NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A STATIC SCHEDULER
FOR THE COMPUTER AIDED PROTOTYPING
SYSTEM:
AN IMPLEMENTATION GUIDE

by

Dorothy M. Janson

March 1988

Thesis Advisor Luqi

Approved for public release; distribution is unlimited.

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified		1b Restrictive Markings	
2a Security Classification Authority		3 Distribution/Availability of Report: Approved for public release: distribution is unlimited.	
2b Declassification Downgrading Schedule			
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)	
6a Name of Performing Organization: Naval Postgraduate School	6b Office Symbol (*if applicable*): 62	7a Name of Monitoring Organization: Naval Postgraduate School	
6c Address (*city, state, and ZIP code*): Monterey, CA 93943-5000		7b Address (*city, state, and ZIP code*): Monterey, CA 93943-5000	
8a Name of Funding/Sponsoring Organization	8b Office Symbol (*if applicable*)	9 Procurement Instrument Identification Number	
8c Address (*city, state, and ZIP code*)		10 Source of Funding Numbers	
		Program Element No	Project No / Task No / Work Unit Accession No

11 Title (*include security classification*) A STATIC SCHEDULER FOR THE COMPUTER AIDED PROTOTYPING SYSTEM: AN IMPLEMENTATION GUIDE

12 Personal Author(s) Dorothy M. Janson

13a Type of Report	13b Time Covered	14 Date of Report (*year, month, day*)	15 Page Count
Master's Thesis	From To	March 1988	125

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (*continue on reverse if necessary and identify by block number*)
Field	Group	Subgroup	rapid prototyping, static scheduler, CAPS, PSDL, Ada

19 Abstract (*continue on reverse if necessary and identify by block number*)

As demand for hard real-time and embedded computer systems increases, a new approach to software development is critical. Software engineers and users would benefit from an automated methodology allowing validation of design specifications or functional requirements early in the development life cycle. A fast, efficient, easy-to-use tool would increase productivity and would enhance user confidence that software would be delivered at less cost and on schedule. The Computer Aided Prototyping System (CAPS) is a conceptualized tool providing these capabilities.

This thesis represents a pioneering effort to develop a Static Scheduler for the CAPS Execution Support System using the Ada® programming language. The Static Scheduler initially extracts critical operators, timing constraints and precedence relationships from a high-level prototype source program. The Static Scheduler then creates a static schedule for run-time execution, using worst case scenarios, guaranteeing that timing constraints are met. The primary goal of this thesis is to provide the scheduling algorithms and implementation guidelines for the Static Scheduler. Secondary goals are to demonstrate the significance of continued research to telecommunications applications and to demonstrate the feasibility of Ada® as the implementation language. (Ada® is a registered trademark of the United States Government, Ada Joint Program Office.)

20 Distribution/Availability of Abstract	21 Abstract Security Classification	
☒ unclassified unlimited ☐ same as report ☐ DTIC users	Unclassified	
22a Name of Responsible Individual: Luqi	22b Telephone (*include Area code*): (408) 646-2735	22c Office Symbol: 52LQ

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

A STATIC SCHEDULER FOR THE COMPUTER AIDED PROTOTYPING SYSTEM: AN IMPLEMENTATION GUIDE

by

Dorothy M. Janson
Lieutenant, United States Navy
B.S., San Francisco State University, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN TELECOMMUNICATIONS SYSTEMS MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 1988

ABSTRACT

As demand for hard real-time and embedded computer systems increases, a new approach to software development is critical. Software engineers and users would benefit from an automated methodology allowing validation of design specifications or functional requirements early in the development life cycle. A fast, efficient, easy-to-use tool would increase productivity and would enhance user confidence that software would be delivered at less cost and on schedule. The Computer Aided Prototyping System (CAPS) is a conceptualized tool providing these capabilities.

This thesis represents a pioneering effort to develop a Static Scheduler for the CAPS Execution Support System using the Ada® programming language. The Static Scheduler initially extracts critical operators, timing constraints and precedence relationships from a high-level prototype source program. The Static Scheduler then creates a static schedule for run-time execution, using worst case scenarios, guaranteeing that timing constraints are met. The primary goal of this thesis is to provide the scheduling algorithms and implementation guidelines for the Static Scheduler. Secondary goals are to demonstrate the significance of continued research to telecommunications applications and to demonstrate the feasibility of Ada® as the implementation language. (Ada® is a registered trademark of the United States Government, Ada Joint Program Office.)

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The Department of Defense (DOD) and the Department of the Navy (DON) allocate billions of dollars each year for initial development or maintenance of progressively more complex weapons and communications systems. These advanced systems increasingly rely on requirements for hard real-time processing of information, utilizing embedded computer systems to monitor or control system performance. These embedded systems currently perform time-critical functions that are primarily computational in nature, such as missile guidance or communications network control. As early as 1973, studies showed that computer software alone comprised approximately 46 percent (over 3 billion dollars) of the estimated total DOD computer costs of 7.5 billion dollars. In addition, 56 percent of these software costs were devoted specifically to embedded systems. [Ref. 1: p. 14]

As they approach the 21st century, the DOD and the DON will be faced with ever increasing demands for complex, hard real-time or embedded systems. This growth of and dependency on embedded systems is readily apparent when compared with the growing civilian reliance on similar, small-scale systems to prepare their food and "drive" their automobiles. Considering the growth of software development and maintenance costs versus computer hardware costs, users must insist that systems are received on schedule and are responsive to stated needs; that they are reliable, efficient and within cost estimates; and, that they are modifiable and transportable to other applications. Fulfilling these user demands requires a systematic approach to software development and an ability to deal with complex solutions.

1. Software Engineering

Software engineering is the application of science and mathematics (specifically algorithms) by which the capabilities of computers are made useful through the application of computer software programs, procedures and related documentation. Generally accepted goals for software engineering fall under the two related categories of performance and design quality. System performance is of primary importance to the user. A delivered software system must accurately represent the user's stated requirements and also consistently produce highly reliable responses within the anticipated environment. Quality of the design is becoming increasingly important to both the user

and the systems engineer. A delivered system must be efficient in its use of resources, understandable, and modifiable in order to minimize modification or maintenance costs in the future. These goals are achievable by utilizing a modular architecture, localization of logically-related resources, uniformity of notation, accuracy of the minimum required elements and confirmability through the use of demonstrations. [Ref. 1: pp. 29-35] Software engineering encourages the use of a development life cycle methodology that systematically and consistently incorporates these goals and principles in the creation of software systems.

2. Traditional Life Cycle

The traditional waterfall life cycle methodology currently used for developing large software programs incorporates individual development stages. These stages include requirements analysis, functional specifications, architectural design, module design, implementation and testing. Requirements analysis establishes the purpose of the intended system and defines the currently perceived constraints or boundaries within which the system will function. The functional specifications define aspects and interfaces of the proposed system that are visible to the user. The architectural design describes a high-level model of the system while module design describes the algorithms and data structures for implementing the architectural design. The implementation stage involves the actual hand coding of the executable programs in a programming language which are then tested for performance accuracy. These stages are normally completed individually and sequentially before system performance is validated by the user. Inconsistencies with expected performance could result in minor changes to software implementation or in drastic modifications if design misconceptions occurred during the requirements analysis or functional specification stages. Depending on the full impact of these inconsistencies, delivery dates could slip, costs in dollars and man-hours could increase, and the overall reliability and accuracy of the software product could be in question.

When this traditional life cycle approach is applied to hard real-time or embedded systems, the potential for major inconsistencies increases. One of the major differences between a real-time system and a conventional computer system is the required precision and accuracy of the application software. The response time of each individual operation may have a unique value to the system which varies with time. In hard real-time systems this response time is a critical determining factor in the accuracy of the software. These response times, or deadlines, must be met or the system will fail to

function correctly, often leading to catastrophic consequences. [Ref. 2: p. 113] For example, as part of a larger computer system, an embedded system can incorporate stringent real-time constraints, parallel processing using two or more computers, and a high degree of reliability. These requirements will often exceed the capacity or capabilities of one software engineer, but may instead require several individuals working independently on different segments of the system. In summation, without the aid of automated software tools, development of hard real-time systems using a traditional life cycle approach can become inefficient and less effective.

3. Rapid Prototyping

a. Conventional Rapid Prototyping

Current research suggests a revised methodology for the software development life cycle, especially when designing hard real-time systems, which consists of two phases, rapid prototyping and automatic program generation [Ref. 3: p. 2]. Although current capabilities preclude completely automatic program generation, the required software tools and capabilities do exist for rapid prototyping. As a software methodology, rapid prototyping provides the user and designer with a fast, efficient and easy-to-use stepwise process. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specifications and initial design before valuable time and effort are expended on implementation software.

Figure 1 on page 4 graphically describes this methodology as a typical feedback loop [Ref. 4: p. 3]. Rapid prototyping initially establishes an iterative process between the user and the designer to concurrently define specifications and requirements for the time critical aspects of the envisioned system. The designer then constructs a model or prototype of the system in a high-level, prototype description language. This prototype is a partial representation of the system, including only those critical attributes necessary for meeting user requirements, and is used as an aid in analysis and design rather than as production software. [Ref. 4: pp. 2-5] During demonstrations of the prototype, the user validates the prototype's actual performance against its expected performance. If the prototype fails to execute properly or to meet any critical timing constraints, the user identifies required modifications and redefines the critical specifications and requirements. This process continues until the user determines that the prototype successfully meets the time critical aspects of the envisioned system. Following this final validation, the designer uses the validated requirements as a basis for the design and eventual hand coding of the production software.

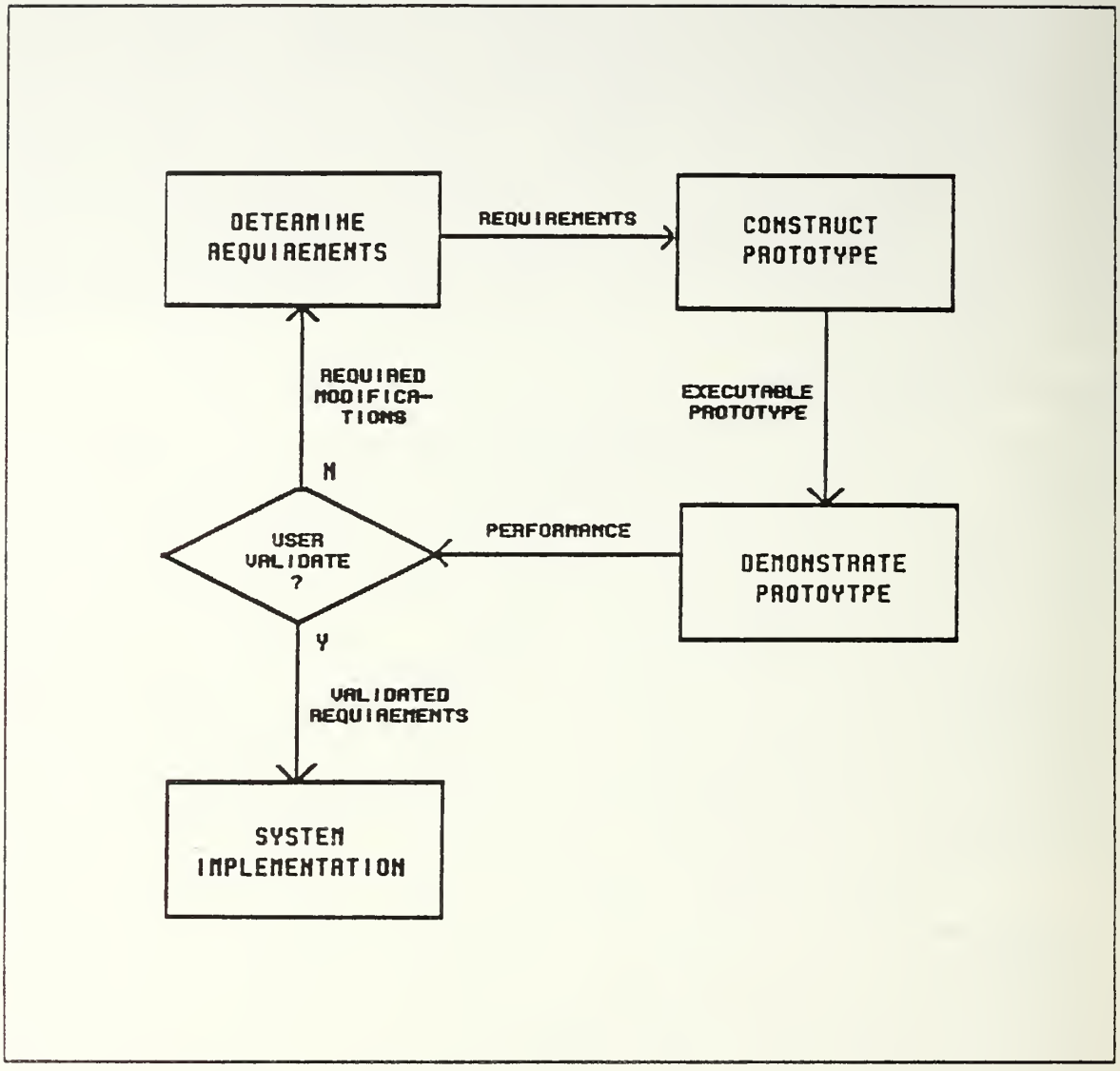


Figure 1. Rapid Prototyping Method

b. Computer-Aided Rapid Prototyping

Computer-aided rapid prototyping further refines the efficiency and accuracy of this new methodology. While utilizing the same iterative approach, computer-aided rapid prototyping relies on three major software tools which assist the designer in constructing and executing the prototype. First, the computer-aided environment includes a software base management system which creates uniform retrieval specifications for software modules in the software database and later retrieves these reusable modules for assembling the executable prototype. Second, a graphics-capable user

interface including a syntax-directed editor expedites the designer's data entry at a terminal and prevents syntax errors in the design. Finally, an execution support system demonstrates and measures the prototype's performance and analyzes the accuracy of design specifications. [Ref. 5: p. 4] Chapter II of this thesis describes the first two tools in more detail while Chapter III provides a detailed description of the third tool.

A computer-aided rapid prototyping approach will provide the software designer with a powerful tool, designed specifically for development of hard real-time or embedded systems. Although the traditional approach may also produce an acceptable product, rapid prototyping suggests significant advantages in several major areas. Designing a simplified executable prototype of the envisioned system forces the user and the designer to decompose a complex system into its major components. This process creates modules that individuals can easily understand and manage. This modularized design is enforced by a formal prototyping language based on abstractions of the system's requirements and high-level constructs of the language itself. A computer-aided rapid prototyping approach using a modularized design focuses the designer's attention on analysis of the requirements and specifications of the system. At this stage, the designer should concern himself with the architectural decomposition of a complex system rather than becoming engrossed with detailed programming efforts inherent in conventional prototyping. This approach allows the user to verify requirements and to identify problem areas early in the development cycle. This verification process eliminates expensive redesign efforts and increases the user's confidence that the system, as envisioned, is feasible. [Ref. 3: pp. 2-3]

Rapid prototyping offers promising advantages in improved software engineering productivity, increased reliability of the finished product, more realistic cost estimates based on identified system complexity, and a reduction in the total conception to operational timeframe [Ref. 4: pp. 11-12]. Ongoing research and pioneering efforts must now fully elevate computer-aided rapid prototyping from its conceptualized design into a functioning reality.

B. OBJECTIVES

This thesis describes the design and implementation efforts for the Static Scheduler subsystem of the Execution Support System (ESS) for the Computer Aided Prototyping System (CAPS). The primary objective of this thesis is to present the algorithms which successfully schedule the critical time constraints in a hard real-time or embedded system model and to establish implementation guidelines for the Static Scheduler. In achieving

this objective, this thesis will also document the Ada®[1] programming language constructs utilized in developing the Static Scheduler.

C. ORGANIZATION

Chapter II describes the external rapid prototyping environment where a Static Scheduler is utilized to execute the prototype of a hard real-time system. This environment includes the CAPS and the Prototype System Description Language (PSDL). This Chapter also presents the software tools used in developing the Static Scheduler. Chapter III concentrates on the CAPS Execution Support System and, specifically, its Static Scheduler. The interfaces between and responsibilities of the Dynamic Scheduler, Translator and Static Scheduler subsystems are outlined, stressing the importance of the Static Scheduler in ensuring accurate execution of critical timing constraints. Chapter IV outlines the analysis and programming decisions that were made or encountered during development of these guidelines. Chapter V presents an application of computer-aided rapid prototyping to a telecommunications switching system. Chapter VI presents conclusions and recommendations stemming from this pioneering research effort.

---

1 Ada® is a registered trademark of the United States Government, Ada Joint Program Office.

II. EXTERNAL ENVIRONMENT

A. ELEMENTS OF THE EXTERNAL ENVIRONMENT

Computer-aided rapid prototyping and its revised outlook on the development life cycle is a familiar topic with software engineering and development professionals. But to the new entrant or to procurement liaison personnel, a basic appreciation for the complexity of creating prototypes for hard real-time systems would prove beneficial. This Chapter, therefore, covers the major areas which affect or help create the environment within which the Static Scheduler operates. First, a description of the CAPS and PSDL provides the reader with a greater understanding of the contribution that the Static Scheduler makes to prototype development. The Chapter concludes with a description of the Kodiyak translator generator, the Ada® programming language, and the hard real-time system model that were used in implementing the Static Scheduler.

B. COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)

The computer-aided rapid prototyping tool addressed in this thesis which incorporates a Static Scheduler is the CAPS. Recognizing that available prototyping methodologies require extensive amounts of individual time and effort, CAPS is designed specifically as a development tool for hard real-time systems. Its primary objective is development of a specification method for identifying and later retrieving reusable software components from an online database while utilizing a formal prototyping language. Together with an iterative process similar in concept to Figure 1 on page 4, CAPS will provide an effective and efficient tool for constructing and validating a prototype. [Ref. 4: pp. 1-2] Rapid construction of this prototype relies on the applicable prototyping method and on a support environment which automates the steps involved. The following sections and Figure 2 on page 8 describe the components of the CAPS architecture and how they work together to make computer-aided rapid prototyping possible.

1. Components of CAPS

CAPS is initialized through the User Interface (UI) as the user and designer work together in defining the critical attributes of the envisioned system. The UI consists of a syntax-directed editor for the formal prototyping language and a graphics tool for displaying data flow diagrams. The editor eliminates syntax errors by prompting the designers with appropriate alternatives at each step of the design process. The graphics

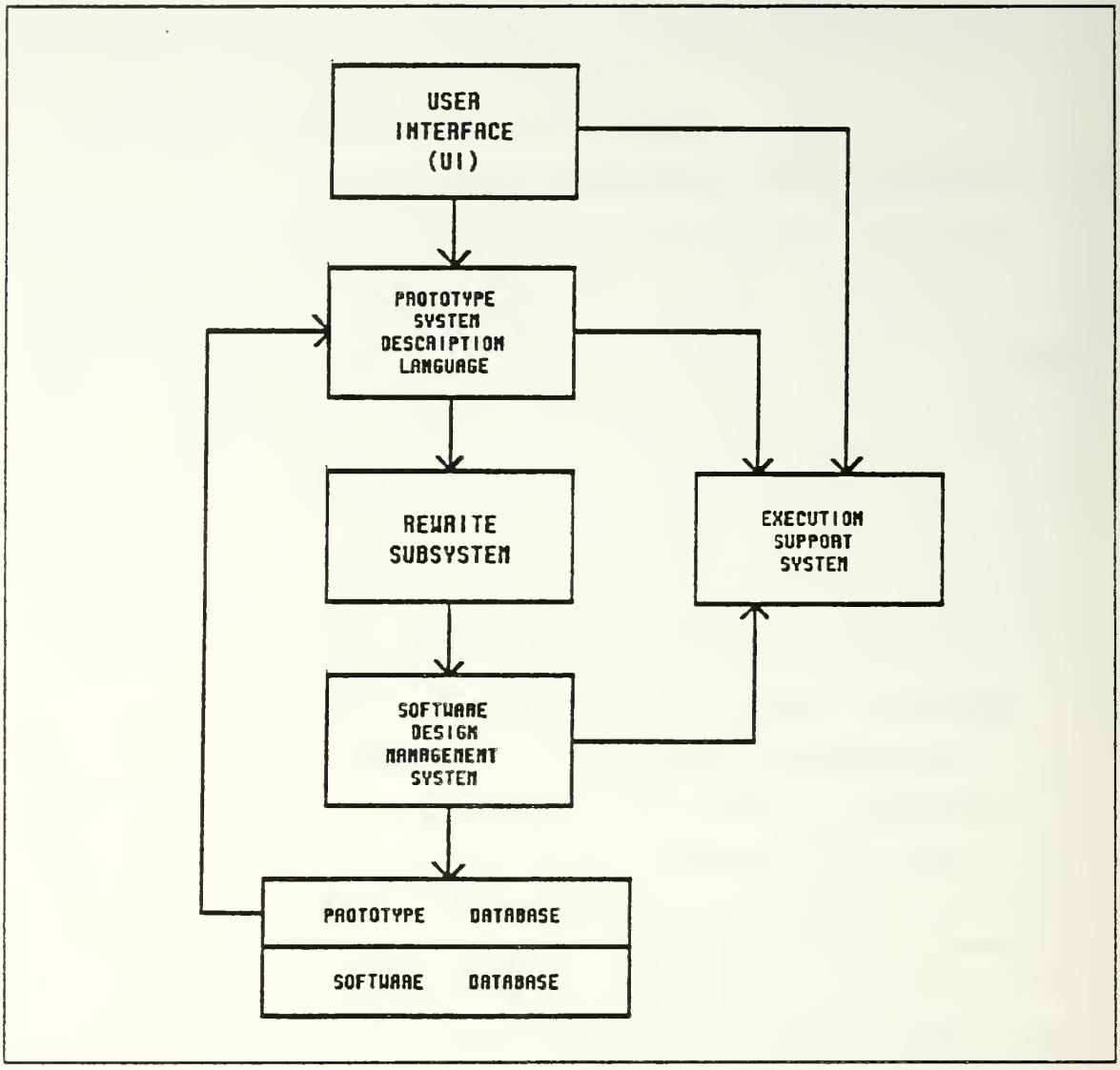


Figure 2. CAPS Architecture

tool provides a picture of the data flow diagrams which reinforces the verbal description of the system specifications. [Ref. 6: pp. 6-7]

The Prototype System Description Language (PSDL) was designed as the primary connection between the designer and the remaining components of CAPS. By definition, PSDL is a high-level prototyping language with special features appropriate for defining critical real-time constraints and is applied at the specification or design stage. [Ref. 7: pp. 3, 23] In order to rapidly construct a prototype, PSDL also includes its own associated prototyping method and automated support environment.

Using a top-down design approach, the PSDL method aids the designer in systematically refining and decomposing each critical component into its lower level components. Uniform PSDL specifications associated with each lower level description act as templates for retrieving reusable software components having similar specifications from the CAPS Software Database. Thus, the PSDL method produces a computational model consisting of the basic building blocks needed to describe the abstractions and concepts of the hierarchically structured prototype. The PSDL execution support environment then verifies the design and the validity of the prototype's real-time requirements. The actual execution of the prototype demonstrates whether these critical timing constraints will perform in an acceptable manner that meets the timing constraints of the system as a whole [Ref. 6: pp. 2-7]. A detailed description of PSDL appears later in this chapter.

The CAPS Rewrite Subsystem provides a means for automatically generating uniform specifications for each reusable software module in the CAPS Software Database and for each PSDL lower level component. Existing methods for retrieving reusable modules are based on keywords. The Rewrite Subsystem, however, uses an approach which allows the designer to give more precise PSDL specifications. The Rewrite Subsystem then transforms each specification into a uniform or normal form. This transformation is achieved by mapping a semantic alias to its normalized term as shown in Figure 3 on page 10. [Ref. 3: pp. 7-8] These normalized specifications are free from ambiguity and create a template used by the Software Design Management System (SDMS) to retrieve software modules from the database with corresponding normalized specifications [Ref. 8: pp. 3-10].

The SDMS is similar to a database management system with additional features required for computer-aided design applications. The SDMS is responsible for organizing, retrieving and instantiating the reusable software modules from the CAPS Database. Retrieval of reusable modules is supported by the normalized specification templates described above. The SDMS instantiates these modules as specified by the designer for execution of the current PSDL prototype. Overall, the SDMS supports efficient selection and retrieval of the relevant software modules. [Ref. 3: p. 9] This minimizes instances where a non-match requires manual creation of a new software module before execution of the prototype occurs.

Two distinct subdivisions of the CAPS Database are the Prototype Database and the Software Database. The first maintains and manages the PSDL prototypes along with their sets of requirements. It also records successive refinements of the

TERM	ALIASES
READ	GET, FETCH, OBTAIN, INPUT, RETRIEVE
UPDATE	CHANGE, MODIFY, REFRESH, REPLACE

Replace all occurences of an alias with its term.

Example: "RETRIEVE temperature from thermometer and
REFRESH the temperature_reading"

is normalized to

"READ temperature from thermometer and
UPDATE temperature_reading"

Figure 3. Normalizing a Specification

prototype alternatives. This process includes facilities for backtracking to previous versions or for combining successive design decisions from the different alternatives. [Ref. 5: p. 10] The Software Database contains the reusable software modules together with their PSDL normalized specifications. This database allows future growth as new modules are identified for inclusion into the database.

The ESS, although a component of the CAPS architecture, actually provides the execution support environment for the PSDL prototyping method. After assembling a prototype of the envisioned system, the three ESS subsystems provide the capability to demonstrate the prototype's actual performance. One subsystem, the Static Scheduler, reads the PSDL prototype source program to identify and extract the PSDL operators with their timing constraints and precedence relationships. For operators with critical timing constraints, the Static Scheduler creates a static schedule, if a feasible one exists, guaranteeing their accurate execution using worst case time scenarios. A second subsystem, the Translator, also reads and translates the PSDL prototype source program

to augment implementation of atomic operators and data types with software code. The Translator accomplishes this by communicating the PSDL prototype's specifications to the SDMS and assembling the executable prototype from the reusable software modules. The final subsystem, the Dynamic Scheduler, maintains run-time execution control of the prototype using the completed static schedule. The Dynamic Scheduler must also schedule operators without critical timing constraints during any excess time slots that remain after each time critical operator completes execution. [Ref. 4: pp. 6-7] These subsystems, and specifically the Static Scheduler, are described in detail in Chapter III of this thesis.

2. Prototype Development with CAPS

Development of an executable prototype with CAPS requires an improved modular design which supports retrieval of reusable software modules and an automated support environment which minimizes the designer's manual involvement in searching for and retrieving the appropriate software modules. The individual CAPS subsystems that provide these automated capabilities are shown in Figure 2 on page 8 and are described in the previous section. Figure 4 on page 12 illustrates the process that the designer uses to interact with the CAPS to develop a prototype.

Initially, the user and the designer jointly determine the specifications of the envisioned software system. The Rewrite System then normalizes these specifications in order to formulate the database queries. The SDMS utilizes these normalized queries to search the Software Database for reusable modules with matching normalized specifications. This search results in either a single match, multiple matches, or no match. Assuming only one match, the SDMS retrieves the applicable module. When multiple matches occur, the designer manually intervenes to select the most appropriate module for this prototype. After all required modules are similarly retrieved, they are assembled to create the executable prototype.

In cases where no match occurs, the designer can either hand code a new module or decompose the PSDL component. The first option generates a unique module required for this prototype. When the current development iteration is complete, this new module is included in the CAPS Software Database. The second option requires manual decomposition of the composite component into atomic lower level PSDL components. After establishing individual specifications, the designer reenters each new component into the development process starting with the Rewrite System. The system

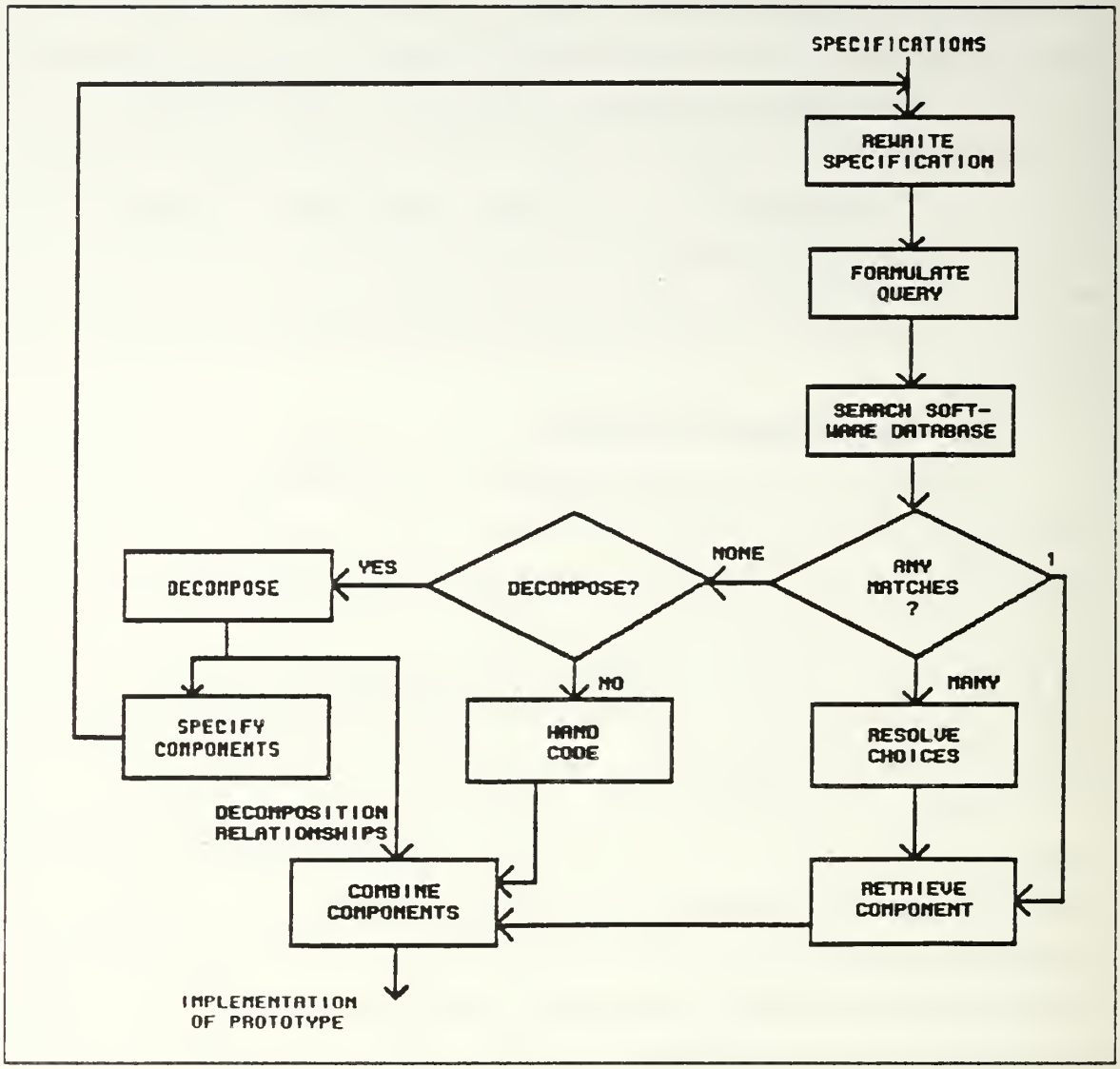


Figure 4. Prototype Development with CAPS

also retains the relationships between decompositions to insure accurate assembly of the retrieved modules during the final stage. [Ref. 3: pp. 4, 16]

3. Prototype System Description Language (PSDL)

The design of PSDL as a high-level prototyping language associated with a rapid prototyping method was specifically influenced by requirements for representing complex, real-time systems. In order to produce a reasonable prototype, PSDL includes the following design requirements:

1. PSDL language and method are simple and easy to use.

2. PSDL creates an executable prototype.
3. PSDL supports a hierarchically structured prototype which simplifies the prototyping process.
4. PSDL supports retrieval of reusable modules from a database using precise specifications.
5. PSDL uses a computational model that explicitly identifies interactions between components which encourages modularization.
6. PSDL contains functional, data and control abstractions for representing the critical timing constraints of operators. [Ref. 6: p. 10]

The PSDL prototyping method encourages a top-down design strategy that focuses the designer's attention on the critical areas or attributes only of the system. An iterative process decomposes and refines these critical elements, using PSDL abstractions to hide lower level programming details. Working together within the execution support environment, the PSDL language and method produce a modularized computational model that contains the necessary timing constraints to simulate the critical portions of the envisioned system. The following sections explain the structure of the computational model and describe the use of PSDL abstractions. Appendix A includes a complete listing of the PSDL grammar.

a. PSDL Computational Model

The PSDL language constructs and modular design rely on a computational model of the system which contains operators communicating via data streams. Mathematically, the computational model is an augmented graph such that

$$G = (V, E, T(v), C(v))$$

where V equals the set of vertices (operators), E equals the set of edges (data streams), T(v) equals the maximum execution time for each vertex (v), and C(v) equals the set of control constraints for each vertex (v). Using the above components, the PSDL enhanced data flow diagram represents a directed graph of the critical aspects of the envisioned software system, including both the timing and control constraints. [Ref. 9: pp. 8-9] The following sections describe the four basic components of the augmented graph.

(1) Operators. PSDL operators represent either functions or state machines. A PSDL operator fires when triggered by either the arrival of a set of input data values or the arrival of a periodic timing constraint. When an operator fires, it reads

one data value from each input stream and writes, at most, one computed data value onto each output stream. The function operator computes an output value based on the current input data values only. Since the state machine is a composite, cyclical operator, one of its data streams acts as a current state variable and controls the feedback loop of the sub-operators. Thus, the state machine computes an output value based on the current input data values and the current value of its internal state variable.

PSDL operators are also identified as either atomic or composite. Atomic operators represent a single operation and cannot be decomposed further. Composite operators represent a network of data and control flow lower level operators. This network contains implied precedence relationships between the sub-operators such that

> if the output from operator A is input to operator B,
> then operator A must fire before operator B.

Thus, a composite operator is decomposed into lower level representations while maintaining the required precedence relationships. [Ref. 9: p. 9]

(2) Data Streams. PSDL data streams are communication links between two PSDL operators, the producer (output) and the consumer (input). Each stream represents a sequential flow of data values such that

> if data value A is generated before data value B,
> then A must be delivered to the next operator before B.

This "pipeline" ordering of operators is required for real-time computations [Ref. 10: p. 127].

These data streams are designed as either data flow or sampled streams. A data flow stream, similar to a discrete data flow, guarantees that each data value on the stream is delivered and acted upon only once. The data value computed by the operator thus represents a unique operation. Data flow guarantees that data values are not lost or repeated by utilizing a first-in first-out (FIFO) queue. This format enforces strict timing relationships between the producer and consumer to insure that the queue does not overflow and that the consumer is not required to wait for input data values.

A sampled stream, similar to a continuous flow, guarantees that data values can be entered onto or delivered from a data stream as they are required by the operators. A sampled stream does not require protection against lost or repeated values since only the most recent value is of interest. A sampled stream utilizes a single memory cell which is updated whenever the producer generates a new data value. This format does not require strict timing between the operators since a producer can generate new values whenever or as often as the consumer requires. [Ref. 9: pp. 10-12]

(3) Timing and Control Constraints. When considering prototypes for hard real-time systems, the timing and control constraints for operator firing and execution are critical. Within the computational model, each time critical operator includes a maximum execution time which gives the worst case time to complete execution once the operator fires. Critical operators can also include conditional control constraints that act as guards. These guards stipulate firing conditions for an operator, conditions necessary before computed values are output onto data streams, or exception situations. [Ref. 6: p. 25]

b. PSDL Abstractions

The PSDL language supports three types of abstractions that assist in developing a simplified but flexible model of the critical properties of a complex system. The operator, data type and control abstractions represent the major building blocks for constructing the PSDL prototype.

(1) Operator Abstractions. PSDL operator abstractions represent either functional or state machines. Each operator has a PSDL specification and implementation section. The specification section contains attributes which describe interfaces to other PSDL components, formal and informal descriptions of the operator's observable behavior, and information required for creating queries to retrieve the reusable modules from the software database. Specifically, each set of attributes contains generic parameters, input, output, states, exception and timing information as shown in Figure 5 on page 16. Only state machine abstractions include the states information, which identifies the state variable and assigns its initial value.

The operator implementation section indicates whether an operator abstraction is atomic or composite. An implementation for an atomic abstraction contains a keyword which specifies the programming language and the name of the retrieved reusable module that implements this operator. An implementation for a composite abstraction contains a set of attributes which includes graph, internal data, control

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart:  medical_history,
     treatment_switch:  boolean
 OUTPUT treatment_finished:  boolean
 STATES temperature:  real
  INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END
```

Figure 5. PSDL Operator Specification

constraints and an informal description of the operator's observable behavior. Figure 6 on page 17 provides an example of a composite abstraction. The PSDL graph utilizes an enhanced data flow diagram which is based on the PSDL computational model's augmented graph. The PSDL graph combines operator specification and implementation information to graphically describe the operator's abstractions and interfaces. [Ref. 9: pp. 14-15]

(2) Data Type Abstractions. PSDL data type abstractions provide a means to distinguish between the prototype's partial representation of critical operators and the operators' actual behavior in the envisioned system as a whole. The PSDL prototype language enforces strong typing by requiring that both pre-defined and user-defined data types be immutable. This rule encourages the designer to clearly and explicitly define an operator's type within the context of the prototype, ignoring unnecessary details. Possible data types include the subset of Ada® pre-defined types, user-defined abstract types, PSDL type constructs, and the PSDL special types TIME and EXCEPTION. These immutable data types result in two important considerations when developing prototypes for complex systems:

- No implicit communication occurs between operators.
- Common interfaces improve comparison with the envisioned system during validation of the prototype.

Each data type has a PSDL specification and implementation section similar in content and format to the operator abstraction. [Ref. 9: pp. 13-16]

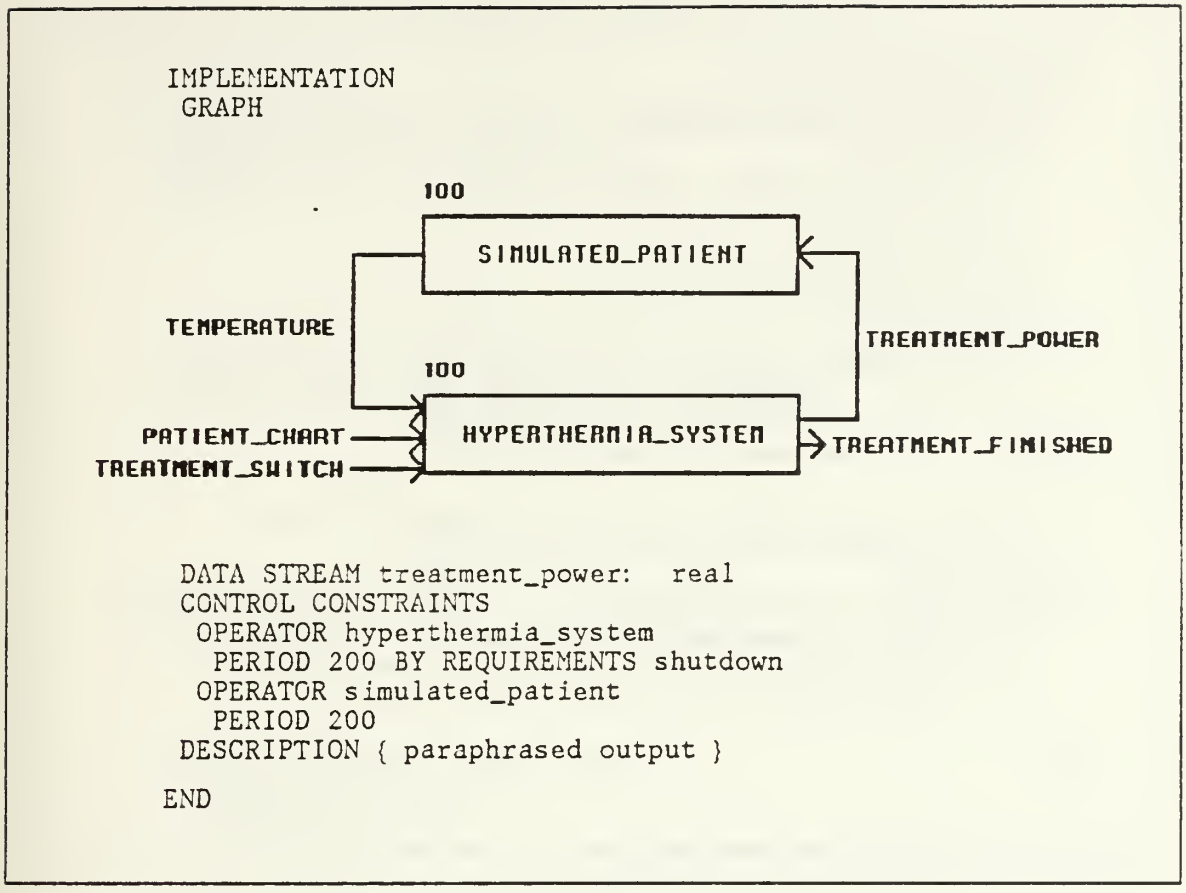


Figure 6. PSDL Operator Implementation

Figure 7 on page 18 and Figure 8 on page 19 provide examples of a data type specification and implementation.

(3) Control Abstractions. PSDL control abstractions provide a means to explicitly describe the periodic execution of operators. These abstractions are represented as a set of control constraints within the PSDL implementation graph of each operator. The actual or scheduled order of operator execution is determined by the Static Scheduler. The Scheduler utilizes these constraints to recognize the precedence relationships between the enhanced data flow diagrams of the operators. Control constraints include data triggers, periods, conditionals, TIMER and EXCEPTION.

The primary control constraints that affect all PSDL operators are the data trigger and the operator's period. All operators must have either a period or a data trigger, or both. The period indicates periodicity of execution for an operator while a data trigger indicates the firing frequency of an operator. Firing frequency can be

```
TYPE medical_history
SPECIFICATION
 OPERATOR get_tumor_diameter
  SPECIFICATION
   INPUT patient_chart:  medical_history,
      tumor_location:  string
   OUTPUT diameter:  real
   EXCEPTIONS no_tumor
   MAXIMUM EXECUTION TIME 5 ms
   DESCRIPTION
   { Returns the diameter of the tumor at a given location,
    produces an exception if no tumor is at that location.
   }
 END


 KEYWORDS patient_charts, medical_records, treatment_recor
    lab_records
 DESCRIPTION
 { The medical history contains all of the disease and
   treatment information for one patient.  The operations
   for adding and retrieving information not needed by
   the hyperthermia system are not shown here.
 }

END
```

Figure 7. PSDL Data Type Specification

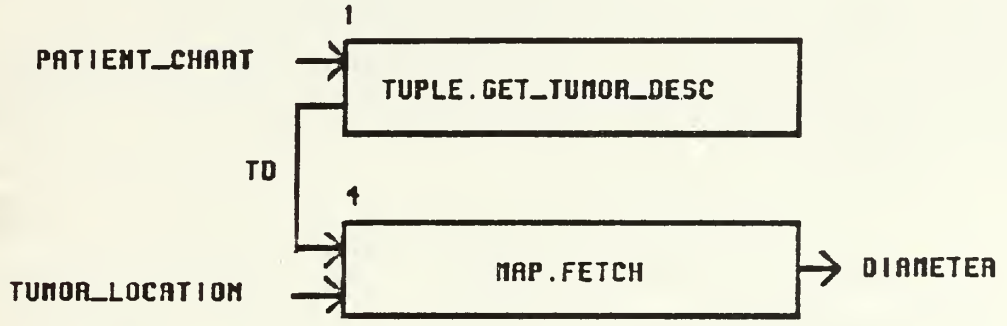
either periodic (synchronous) or sporadic (asynchronous). Periodic operators are triggered at approximately regular intervals which insures that execution is completed between the beginning of each period and its deadline, which defaults to the end of the period. Sporadic operators are implemented by their periodic equivalents which are triggered by the arrival of new data values. The following examples illustrate the PSDL data triggers and their interpretations:

- OPERATOR s TRIGGERED BY ALL x, y, z
- OPERATOR q TRIGGERED BY SOME a, b.

The first example means that s is ready to fire whenever new data values arrive on all three input streams x, y and z. The second example means that q is ready to fire when any one of the data values a or b arrive.

Conditional constraints describe the execution and transmission requirements for the input and output data streams of an operator. Conditional execution

```
IMPLEMENTATION
 tuple[tumor_desc: map[from: string, to: real], ... ]
 OPERATOR get_tumor_diameter
  IMPLEMENTATION
   GRAPH
```




```
    DATA STREAM td:  tumor_descr
    CONTROL CONSTRAINTS
     OPERATOR map.fetch
      EXCEPTIONS no_tumor IF not(map.has(tumor_location, td))

   END_
```

Figure 8. PSDL Data Type Implementation

of an operator requires an IF predicate along with the data trigger. This conditional predicate must be satisfied before the operator is triggered and execution begins. Conditional transmission of an operator's output requires an IF predicate. This condition must be satisfied before an operator can output a data value. The following examples illustrate how these constraints appear in the PSDL implementations:

- OPERATOR x TRIGGERED IF y : critical
- OPERATOR x OUTPUT z IF 1 < z and z < max

where x is the operator_id, y is the input data stream value and z is the output data stream value.

The TIMER represents a unique type of state machine which functions similar to a stopwatch. An operator TIMER can be used to record the length of

time between events or the length of time spent in a given state. Control of the TIMER is local to the component, whether atomic or composite, in which it is declared. Only the TIMER value can be transmitted via a data stream to outside of the local component. TIMER utilizes four primitive operations which include READ the current value, START the timer, STOP the timer and RESET the timer with a value equal to zero.

EXCEPTION represents unique situations that are either system or user-defined. The underlying operating system raises a system-defined EXCEPTION when extraordinary situations preclude continued execution of the program. A user-defined EXCEPTION alerts the prototype demonstrator to situations that indicate critical timing or control constraints were not satisfied during execution. For example, the control constraint

OPERATOR d EXCEPTION f IF e > 10

transmits the EXCEPTION named f on the output streams of d instead of the value computed by d if the value of e is greater than 10. [Ref. 9: pp. 16-21]

(4) PSDL Timing Constraints. Hard real-time systems differ from historical systems primarily due to timing constraints which control proper execution of the system. Control in this context refers to the critical timing relationships between the operators. The three fundamental timing constraints are maximum execution time (MET), maximum response time (MRT), and minimum calling period (MCP). The designer specifies these critical constraints, using worst case time scenarios, in the PSDL specification and implementation sections of the PSDL module.

Within the PSDL specification section the designer specifies one or more of the above three constraints depending on whether the operator is periodic or sporadic. All time critical operators require an MET which specifies an upper bound on the length of time between that operator's execution initiation and its completion. However, the MET does not include additional time for potential execution scheduling delays. Any operator may also include an MRT constraint whose interpretation differs slightly depending on the type of operator. For periodic operators, MRT specifies the upper bound on the amount of time between the beginning of a period and the instance of time when that operator places the last computed data value onto its output streams during that same period. For sporadic operators, MRT specifies an upper bound on the amount of time between the arrival of a new data value, or set of data values, and the

instance of time when that operator places the last computed data value onto its output streams, in response to the arrival of a new data value or set of data values. In both cases, MRT considers and includes potential execution scheduling delays. Any sporadic operator with an MRT requires an MCP. The MCP specifies a lower bound on the amount of time between the arrival of one set of input data values and the arrival of the next set for that operator. Thus, the MCP indicates the minimum amount of time that must elapse between firings of the sporadic operator.

Within the PSDL implementation section the designer specifies the less straightforward or implicit timing constraints. These constraints include the PSDL constructs for event controlled timers, triggering and output conditions. A PSDL TIMER controls execution of an operator by maintaining the timing functions required to support the conditional specifications for that operator. Triggering and output conditions control operator execution by enforcing conditional requirements which must be met before firing of or output by that operator can occur. [Ref. 9: pp. 23-24]

C. KODIYAK TRANSLATOR GENERATOR

Utilization of CAPS during rapid prototyping of hard real-time systems produces a PSDL prototype source program of the envisioned software system. The ESS's Static Scheduler must identify and extract the critical operators and their associated timing constraints from this PSDL source program before creating the static schedule. The Kodiyak automatic translator generator is the tool which provides the Static Scheduler with the capability to process the PSDL source program. This section begins with a general description of the Kodiyak tool and concludes with its specific application for the Static Scheduler.

Kodiyak is an attribute grammar (AG) based tool which automatically generates a translator. The AG approach provides a way for the designer to assign meanings to each input string in a context-free manner. Thus, the AG-based source code contains application specific grammar and attribute equations written in Kodiyak. The compiled output is a translator in C language. The translator parses each input string and places its translation in a derivation tree whose structure is based on the specified equations. The structure of the tree, therefore, provides the meaning for each string. The three sections of the Kodiyak AG tool are described briefly in the following paragraphs and are illustrated in Figure 9 on page 22. Detailed information on modifying this tool for a specific application are found in References 11 and 12.

```
LEXICAL SCANNER SECTION:

Terminal Symbols                Regular Expressions
-----------------------------   -----------------------------------------
OPERATOR                        :operator OPERATOR
MAX_EXEC_TIME                   :maximum[ ]execution[ ]time
                                |MAXIMUM[ ]EXECUTION[ ]TIME

where "|" equals "or" and "[ ]" equals "any character".


ATTRIBUTE DECLARATIONS SECTION:

Grammar Symbols                 Attributes
-----------------------------   -----------------------------------------
operator                        { trn:   string; } ;
max_exec_time                   { trn:   string; } ;
ID                              { %text: string; } ;

where "trn" equals "translation".


ATTRIBUTE GRAMMAR SECTION:

Grammar Symbols          Syntax { Definition Equations }
-----------------------  -----------------------------------------
max_exec_time            :MAX_EXEC_TIME time
                          { max_exec_time.trn =
                          [ "MET ",i2s(time.trn) ] ; }
                         ;
time                     : INTEGER_LITERAL opt_unit
                          { time.value=s2i(INTEGER_LITERAL.%text)
                                       * opt_unit.value;
                                       time.trn=i2s(time.value);}
                         ;
unit                     :MICROSEC
                          { unit.trn = "1"; }
                         |MS
                          { unit.trn = "1000"; }

                         ;
```

Figure 9. Kodiyak AG Examples

1. Lexical Scanner

As the initial section of the translator, the lexical scanner contains a list of statements which describe each named terminal symbol of the target language as shown in Figure 9 on page 22. The primary function of these statements is to define a set of regular expressions which represent expected text within the input source program. As the translator reads the input program, the lexical scanner sequentially locates regular expressions that match terminal symbols. When a match occurs, the input text is deleted and is replaced with the terminal symbol. [Ref. 11: p. 3]

2. Attribute Declarations

The second section of the translator, the attribute declarations, contains a list of statements which further describes both terminal and non-terminal grammar symbols as shown in Figure 9 on page 22. Each statement names the attribute associated with a grammar symbol and defines the attribute's type, either string or integer. String types have arbitrary length and may be concatenated. The range of integers depends solely on local machine capabilities. Most non-terminal symbols require an attribute declaration while named terminal symbols, such as ID, are included only if necessary for accurate translation. [Ref. 11: p. 7-9]

3. Attribute Grammar

The final section of the translator, the attribute grammar, contains a list of statements as shown in Figure 9 on page 22 which define the syntax and semantics of the target language as a grammar tree. Each statement defines a grammar symbol as a combination of root symbols using associative attribute equations. Items to the left of the colon represent the grammar symbols while expressions to the right of the colon represent the grammar's syntax followed by attribute definition equations in { braces }. In the example used, a translation might be MET 9 MICROSEC. If the input text stated "9 MS", then the translation would be MET 9,000 MICROSEC. [Ref. 11: pp. 9-10]

In this thesis, the Kodiyak tool is specifically tailored as a PSDL processor for the Static Scheduler. The AG source code contains those specific PSDL grammar and attribute equations written in Kodiyak which define the critical operators and their timing constraints and precedence relationships. The compiled C language processor then locates and extracts operators and timing information from the original PSDL source program that are required by the Static Scheduler. Chapter IV contains a

detailed description of the PSDL grammar and attributes that were modified specifically for the Static Scheduler.

D. ADA PROGRAMMING LANGUAGE

From the plethora of computer programming languages available and familiar to software programmers, the designers of CAPS selected Ada® as the most appropriate programming language. Two factors reinforced their selection:

1. DOD's efforts to standardize software, and
2. Ada®'s constructs for representing complex and embedded software systems.

With the increased demand for complex real-time systems, the DOD recognized that the design and management of these systems required a new development approach. DOD initiated development of a programming language incorporating new software engineering principles and including capabilities designed specifically for complex systems. As a result, implementation of CAPS in Ada® will provide a computer-aided rapid prototyping tool compatible with and directly related to the development of future complex software systems. [Ref. 1: pp. 3-4]

Early Ada® designers recognized that complex software systems required parallel processing, real-time control, exception handling, and unique input/output (I/O) control [Ref. 1: p. 16]. From these basic requirements the designers identified a programming style and its associated language constructs that are fundamental to the development of complex systems containing critical timing constraints. At the highest level, the recommended programming style includes conventions for organizing program units and for naming entities. Proper use of Ada®'s program, task and package units insures a modular design supported by separate compilation of individual units and by data abstractions. The preferred Ada® naming conventions for all user-defined entities create software code that is easily understood and self-documenting. The Ada®programming language includes a pre-defined language environment that contains extensive data types, calendar/timing functions, system exceptions, and several levels of I/O operations. However, Ada®programming principles also stress user-defined data types, exceptions, and I/O operations to insure precise implementations and consistent, self-documenting code.

At a lower level, Ada® constructs for task program units containing rendezvous operations are integral concepts for prototyping real-time systems. The rendezvous constructs ENTRY and ACCEPT provide explicit synchronization between two parallel

tasks, supporting both concurrency of operation and precedence relationships between time critical operators. Instantiation of generic program units provides the necessary environment for creating the prototype's abstract representation of the envisioned system. Chapter IV of this thesis describes the application of these constructs during implementation of the Static Scheduler.

E. HYPERTHERMIA MODEL

During the initial conceptual research for the CAPS [Ref. 6], the hyperthermia system was used as a model for studying a hard real-time system. This model provides a dynamic environment with critical timing constraints, but is also small enough to easily understand and manipulate. The hyperthermia system is described as

> .. a medical device for treating tumors . . . which uses a microwave generator connected to a fine tuner and matching control system. The hyperthermia system uses microwave energy to produce and deliver controlled local therapeutic heating directly to tumors for effective and safe treatment of cancer. A computerized control system adjusts power output automatically to maintain the therapeutic temperature in accordance with the established patient treatment plan. [Ref. 13: p. 11]

For the hyperthermia system, the designer must determine which subsystems and attributes are required by the prototype to accurately simulate the envisioned system. The designer of a prototype would only be concerned with subsystems that receive temperature readings from sensors and that generate commands to modify or terminate the treatment. The following critical requirements [Ref. 13: p. 13] were identified:

1. Power must drop to zero within 300 ms after turning off the system.
2. Temperature must remain between 42.2 and 42.6 degrees centigrade.
3. Temperature must never exceed 42.6 degrees centigrade.
4. System must stabilize within 5 minutes after starting treatment.
5. System must shutdown automatically when the temperature is above 42.4 degrees centigrade for 45 minutes.

The resulting PSDL prototype for the hyperthermia system, based on the above requirements, is presented in Appendix B. Sections from this sample prototype program are used throughout this thesis to clarify an idea or provide examples for the implementation.

III. STATIC SCHEDULER CONCEPTUALIZED

A. INTERNAL ENVIRONMENT

Chapter II of this thesis described CAPS as an effective and efficient tool for creating an executable prototype of a hard real-time system. Rapid construction and validation of a prototype require an execution support environment that automates the many time-consuming tasks involved with developing the design up through analyzing the performance of the prototype. Although the ESS represents only one component of the CAPS prototyping tool, it is the primary factor that differentiates CAPS from manual prototyping tools. The ESS provides automated control and interface capabilities that allow the system to save current state information when run-time discrepancies are noted and to execute modified versions of the prototype. [Ref. 6: p. 7]. These capabilities are essential to realize time and cost savings and to increase user confidence that the system's design is feasible.

The ESS contains a Dynamic Scheduler, a Static Scheduler, and a Translator. As conceptualized by the author for this thesis, Figure 10 on page 27 illustrates the ESS subsystems' external interfaces to other components of CAPS and the interactions within the ESS itself. The ESS utilizes four external interfaces from outside the ESS. Initially, a command from the UI to the Dynamic Scheduler activates the ESS when the designer requests execution of the PSDL prototype. Second, the Translator and the Static Scheduler require access to the PSDL prototype source program created jointly by the designer and the user. Third, the Combiner Linker Exporter (CLE) receives, compiles and links the Ada® source code from the Translator and the Static Scheduler. The executable code generated by the CLE becomes input to the Dynamic Scheduler for run-time execution. The final interface from the Dynamic Scheduler to the UI provides prototype execution statistics, analysis and error message information direct to the designer.

The first objective of this Chapter is a description of the ESS subsystems. A brief introduction to the basic functions of the Dynamic Scheduler and the Translator provides a general survey of the execution environment. Emphasis is placed on the interfaces between each of these subsystems and especially with the Static Scheduler. The primary objective of this Chapter is to provide a detailed description of the Static

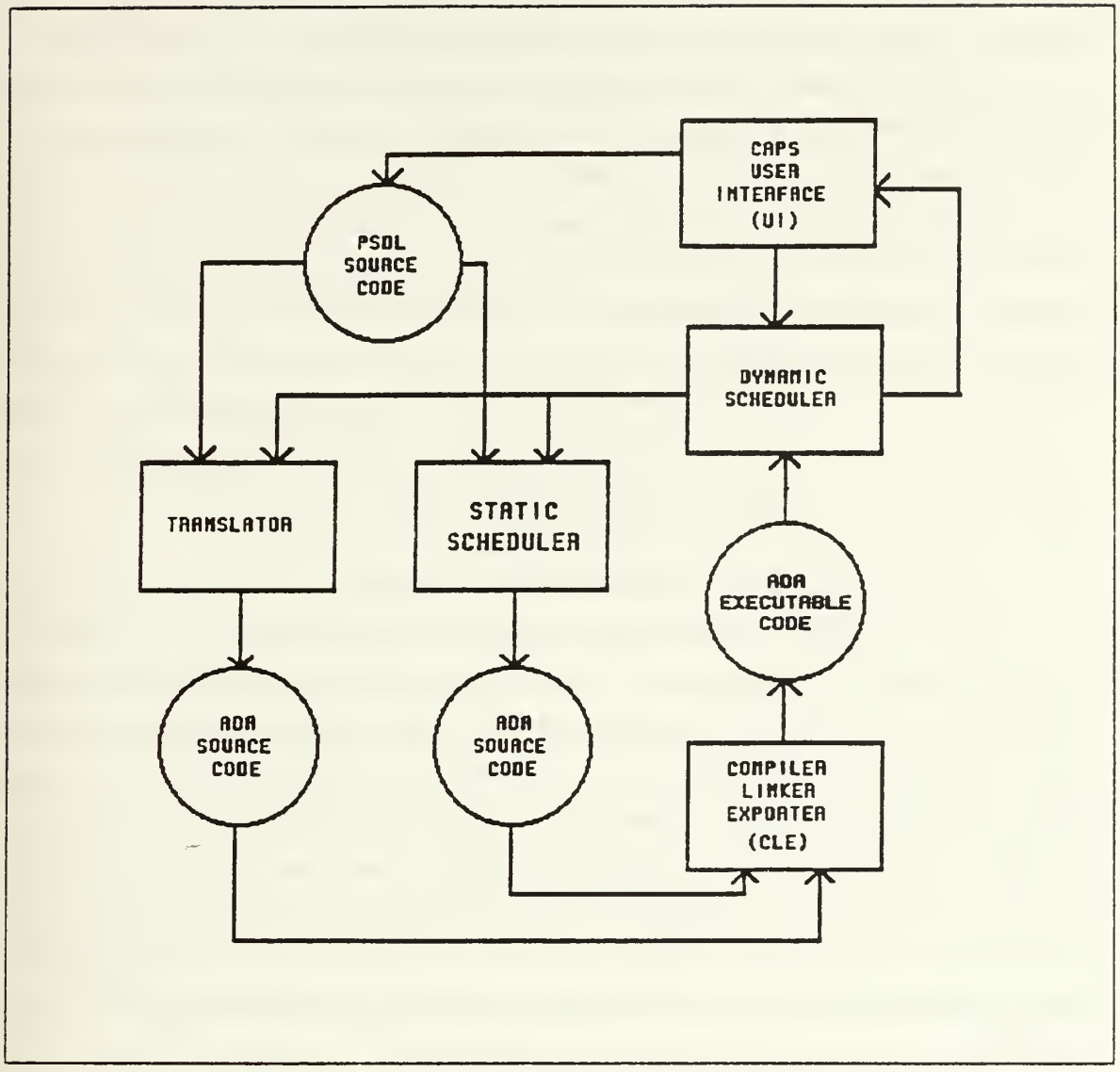


Figure 10. Execution Support System Interfaces

Scheduler. This description covers the functions and responsibilities that the Static Scheduler must accomplish to create a static schedule for time critical operators.

1. Dynamic Scheduler

The conceptualized design for the Dynamic Scheduler utilized in this thesis derives directly from Reference 14 and as initially conceived in Reference 4. The Dynamic Scheduler fulfills two major roles for the CAPS. First, the Dynamic Scheduler exercises the prototype which is a fundamental requirement of a rapid computer-aided prototyping system. Second, prompt feedback from the Dynamic Scheduler to the UI allows the

designer and the user to assess the prototype's execution performance. In order to perform these roles, the Dynamic Scheduler coordinates the functions of the entire ESS.

In its first role, the Dynamic Scheduler acts as the primary link between the CAPS UI and the ESS. When the designer completes the PSDL prototype source program, a request from the UI (or a potential subsystem of the UI) to execute the PSDL prototype is received by the Dynamic Scheduler. The Dynamic Scheduler in turn invokes the Translator and the Static Scheduler, and initiates procedures which pre-load data stream buffers for the Translator. The Translator transforms the PSDL prototype program into an executable Ada® program while the Static Scheduler provides a linear static schedule for time critical operators. An Ada®-compiled implementation of the prototype then becomes an input to the Dynamic Scheduler. After successful completion of these pre-execution functions, the Dynamic Scheduler provides all run-time executive activities while exercising the prototype.

As the ESS driver program, the Dynamic Scheduler has two main responsibilities. First, it schedules operators without timing constraints that were not included in the static schedule. The Dynamic Scheduler receives an input file from the Static Scheduler containing these unscheduled operators. Since the Static Scheduler uses worst case times (METs) for scheduling the critical operators, on the average these operators will not utilize the entire allotted time slots. The Dynamic Scheduler identifies this spare capacity as it occurs and schedules operators without timing constraints during the time remaining. In some cases the operators so scheduled may not complete execution during this time slot. The Dynamic Scheduler must then preempt the operator's execution and abandon the entire operation before the next pre-scheduled critical operator must begin execution.

The second ESS responsibility of the Dynamic Scheduler provides run-time exception handling and hardware operator interrupts. EXCEPTIONs are raised from the Translator as either dataflow overloads when an operation attempts to write to a buffer that is full or dataflow underloads when an operation attempts to read from an empty buffer. EXCEPTIONs are raised from the Static Scheduler when a critical operator exceeds its worst case (MET) time slot or validity checks on constraint values indicate a static schedule is not feasible. In all of these cases, execution of the prototype will stop and an applicable error message is displayed at the UI. The Dynamic Scheduler must also handle conventional interrupts, such as a <cr>C from the user to abort execution or an equipment failure.

Future enhancements identified in addition to the current Dynamic Scheduler design would provide debugging capabilities and statistical information. During execution of the prototype, the debugging capabilities would trace relevant information concerning operator execution. Computed values and their associated input and output times would display a record of events that occur during execution. Statistical information collected during execution would include frequency of operator firing, quantity of EXCEPTIONs occurring, and statistical data on timing parameters for critical operators. [Ref. 4: pp. 10-11] When combined, these two enhancements would provide the designer and the user with precise information for measuring, analyzing and validating the prototype's performance.

2. Translator

The conceptualized design for the Translator utilized in this thesis derives directly from Reference 12 and as initially conceived in Reference 4. The Translator's primary responsibility is the translation of the PSDL prototype source program into an executable Ada® program that simulates the behavior of the prototype. The Translator accomplishes this translation by utilizing a version of the Kodiyak translator generator specifically tailored for this application. The Translator invokes the AG translator program which semantically parses the PSDL program statements into their Ada® program representations. The translator contains a list of PSDL grammar statements with their associated attribute definition equations that represent the corresponding Ada®grammar. These equations define the semantics of the translation using a structured grammar tree approach.

Augmentation code for PSDL atomic operators is embedded within the attribute definition equations. These augmentations implement the PSDL data streams, PSDL operator conditional constraints and PSDL TIMER functions. Both sampled and data flow stream augmentations are implemented with individual buffers containing one computed value for each stream. PSDL operator triggering conditions and output guards are implemented by the equivalent Ada® semantics. A PSDL TIMER is implemented using the CLOCK function from the standard Ada® library package CALENDAR.

During the early prototype design phase, any PSDL composite operators are decomposed into atomic operator networks. Reusable modules from the CAPS Database, considered as Ada® program units for this thesis, are inserted as the implementation code for these operators. The augmentation code described above, combined with

these Ada® implementations, produces the prototype's Ada® source code. The CLE compiles this source code and links it with the compiled static schedule to generate the executable Ada® program. This executable program then becomes the input to the Dynamic Scheduler for execution of the prototype.

B. THE STATIC SCHEDULER

The conceptualized design for the Static Scheduler utilized in this thesis derives directly from Reference 15 and as initially conceived in Reference 4. The primary development emphasis for CAPS was computer-aided rapid prototyping for hard real-time systems. By automating many of the time-consuming tasks of conventional rapid prototyping tools, the ESS and the SDMS differentiate the CAPS from its manual or semi-automated counterparts. But the Static Scheduler subsystem of the ESS alone represents the single most important component of CAPS as the basic requirement for computer-aided rapid prototyping of hard real-time systems. The Static Scheduler specifically addresses only those operators with critical timing constraints whose precise performance determine whether the system, as designed, will meet the required timing specifications.

As conceptualized, the primary purpose of the Static Scheduler is creation of a static schedule which gives the precise execution order and timing of operators with hard real-time constraints in such a manner that all timing constraints are guaranteed to be met [Ref. 4: p. 7]. Assuming that such a schedule is feasible given the system specifications, the static schedule contains the pre-allocated starting time and execution time for each critical operator. This structure implicitly denotes the precedence relationships between the operators. Without the benefit of a Static Scheduler, execution of the prototype would rely on basic control flow and processor scheduling as currently utilized in the majority of software systems. Rapid prototyping in general would benefit from CAPS without a static schedule. However, the Static Scheduler provides CAPS with the unique capability required to realize increased gains in designer productivity and system reliability during development of hard real-time systems.

The remainder of this Chapter provides implementation guidelines describing how the Static Scheduler functions as conceptualized in Reference 15 and by this author. The implementation design in this thesis addresses a single processor application only. The impact on or modification to this design when multi-processors and concurrent processing are utilized will not be addressed explicitly. Data Flow Diagrams (DFDs) in Appendix C illustrate the conceptualized design for implementation of the Static Scheduler. The 1st level DFD presents a general description of the Static Scheduler

while the lower level DFDs contain more specific implementation guidelines. The current discussion addresses the 1st and 2nd level DFDs and outlines assumptions that were made to define the scope of the implementation guidelines.

1. Static Scheduler 1st Level DFD

The 1st Level DFD in Appendix C outlines the five major functions of the conceptualized Static Scheduler [Ref. 15]. The initial input to the Static Scheduler is a text file containing the PSDL prototype program created jointly by the designer and the user. An intermediate output to the Dynamic Scheduler is a text file containing the non-time-critical operators that were extracted from the PSDL program together with the time critical operators. The final output from the Static Scheduler to the CLE is an Ada® source file containing the static schedule. The CLE compiles and links this program to the Translator's compiled Ada® program. This combined program is the executable Ada® program used by the Dynamic Scheduler to demonstrate the prototype's performance. The following sections describe the functions performed by each component of the 1st level DFD and, in the process, provide an introduction to the lower level DFDs.

a. "Read_PSDL"

Following initiation by the Dynamic Scheduler, the Static Scheduler's first major function is reading and processing the PSDL prototype program. Although the Translator performs a similar but extensive process for the entire PSDL program, the Static Scheduler requires only that information which identifies critical operators along with their associated timing constraints and the link statements which syntactically describe the PSDL graphs. A specifically tailored version of the AG-based Kodiyak tool for the Static Scheduler identifies and extracts this information only from the PSDL source program. This process creates a sequential text file containing operator identifiers, timing information and link statements.

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that the PSDL prototype program is syntactically correct. This implies that each line begins with a PSDL keyword or reserved word. Second, this design assumes that the designer structured the PSDL prototype program using a top-down design. This implies that the program begins with the highest level and then decomposes all composite operators, with the last (or lowest) level being the Ada® implementation modules. These assumptions are realistic in that

PSDL encourages the designer's use of a structured, modular architecture and also that the UI will include a syntax-directed editor.

b. "Pre-Process_File"

After the AG processor creates the output text file, the Static Scheduler's next major function is sorting the contents of this file and performing basic validity checks on pertinent information. The input text file is a sequentially ordered file containing all required information as it was extracted from the PSDL program. This information must be divided into three separate files based on its destination or additional processing required. The Non-Crits file contains a sequential list of all non-time-critical operator identifiers which becomes an intermediate input to the Dynamic Scheduler. The Dynamic Scheduler schedules these operators during execution of the prototype as excess time becomes available. The Operator file contains all critical operator identifiers and their associated timing constraints. This file is organized as an array of records. Each record contains fields for the operator identifier and the numeric values of its MET, MRT, MCP, period and FINISH_WITHIN. The Links file contains the link statements which syntactically describe the PSDL implementation graphs. This file is also organized as an array of records. Each record contains fields for the data stream identifier which communicates between the two operators, the producer operator identifier and the numeric value of its MET, and the consumer (user) operator identifier. The derivation of these link statements appears in the section "Sort_Topological".

During this phase, the Static Scheduler also performs basic validity checks on the timing constraints contained in the critical operator file. At a minimum, three validity checks performed at this stage will increase the probability of creating a feasible schedule during the later stages. The first check verifies that an operator record containing an MCP value also contains an MRT value. If the MRT is missing, it is calculated as either (MRT equals FINISH_WITHIN) or (MRT equals MET). A second check verifies that an operator's MET value does not equal its period value, if a period is present in the record. A third check verifies that an operator's MET value is less than its MRT value. [Ref. 16: p. 6] In all three cases, if any one of the checks fails an EXCEPTION will be raised and an appropriate error message submitted to the UI. The significance of these validity checks will become apparent in the section for "Build_Harmonic_Blocks".

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that critical operators will always include an

MET value. If this value is not present, the operator is assumed non-time-critical and is delivered to the Dynamic Scheduler. Second, this design assumes that all timing constraints are non-negative integer values. These assumptions are realistic in that "critical" here implies maximum or minimum timing constraints. In addition, a negative time value would be meaningless and the time units available in PSDL (i.e. mille- or micro-seconds) provide sufficient time divisions.

c. "Sort_Topological"

After the "Pre_Process_File" function creates its output files, the Static Scheduler's next major function is performing a topological sort of the link statements contained in the Links file. In order to appreciate the complexity of this topological sort, Figure 11 on page 34 illustrates a PSDL linear augmented graph and its corresponding sorted link statements. Lower case characters identify data streams which provide data value communication paths between two operators. The upper case characters identify the critical operators. An operator identifier appearing on the left side of the arrow represents a producer of data. An operator identifier appearing on the right side of the arrow represents a consumer (user) of data values. The special word EXTERNAL identifies a situation where the data input/output arrives/exits the current level of the system under consideration depending on whether EXTERNAL is located on the left/right of the link statement. The numerical value on the right side of the colon records the MET value and unit of measurement for the operator identifier on the left side of the colon.

All link statements conform to this same format regardless of the level of decomposition under consideration. However, as these lower levels are encountered, each single statement could be replaced by or affect two or more statements. As an example, Figure 12 on page 35 illustrates the decomposition of operator B from Figure 11 on page 34. A comparison of the two figures indicates that two new statements were added:

- b1' . B1 : 5 ms --> B2
- b2' . B2 : 10 ms --> B3

and two statements were modified:

- b . A : 10 ms --> B is now b . A : 10 ms --> B1
- c . B : 20 ms --> C is now c . B3 : 5 ms --> C.

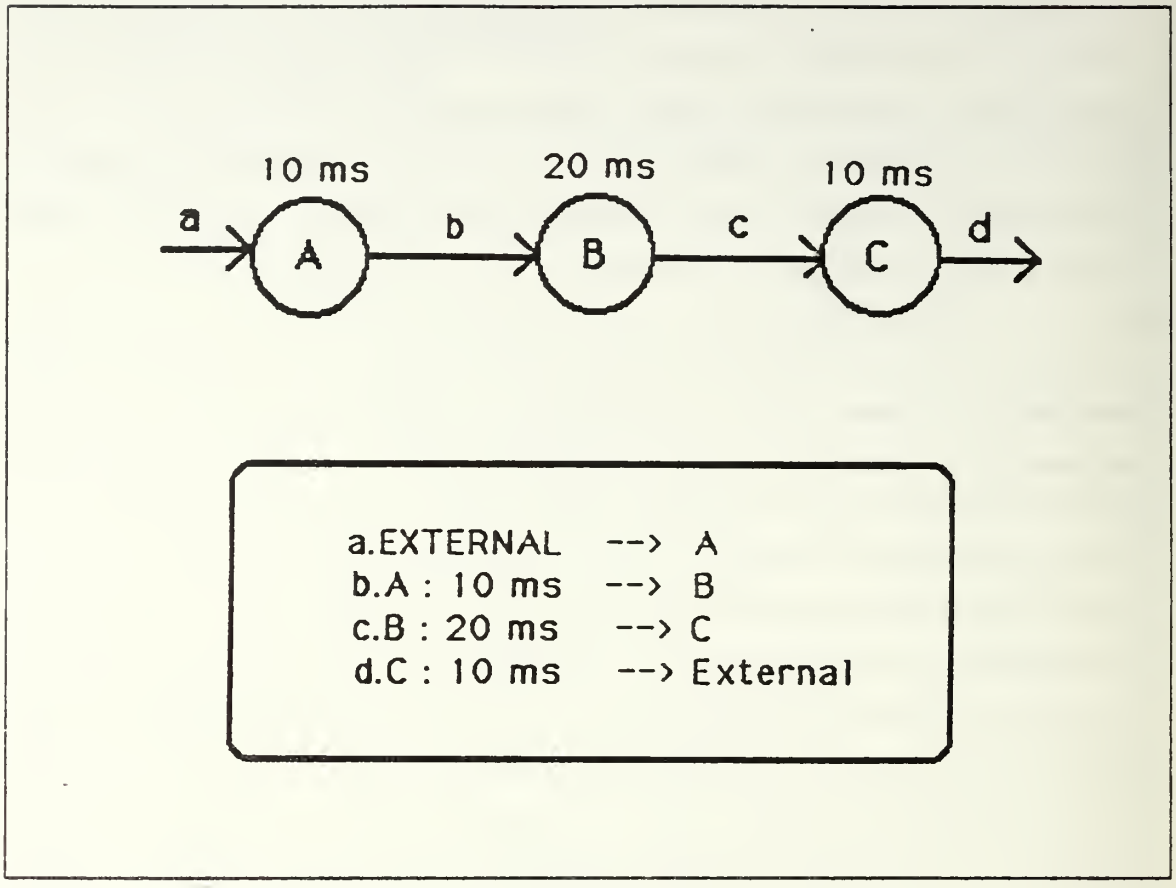


Figure 11. PSDL Graph and Link Statements

All of the link statements from each level appear sequentially in the Links file as they were extracted from the PSDL prototype program.

The requirement for a topological sort implies that the statements being sorted have a natural continuity and connectedness. These properties define the execution precedence of the time critical operators regardless of whether the graphs are linear or acyclic. With a linear graph, the sort establishes a start point by locating the statement containing the EXTERNAL keyword in the left-hand operator position. Conversely, the end point is established by locating a statement containing the EXTERNAL keyword in the right-hand operator position. The remaining operators are ordered by locating matches between the right and left-hand operators. Sorting an acyclic digraph differs only in how the start and end points are established. The acyclic sort establishes a start point by locating the statement(s) having a left-hand operator with no matching right-hand operator. The end point is established by locating the

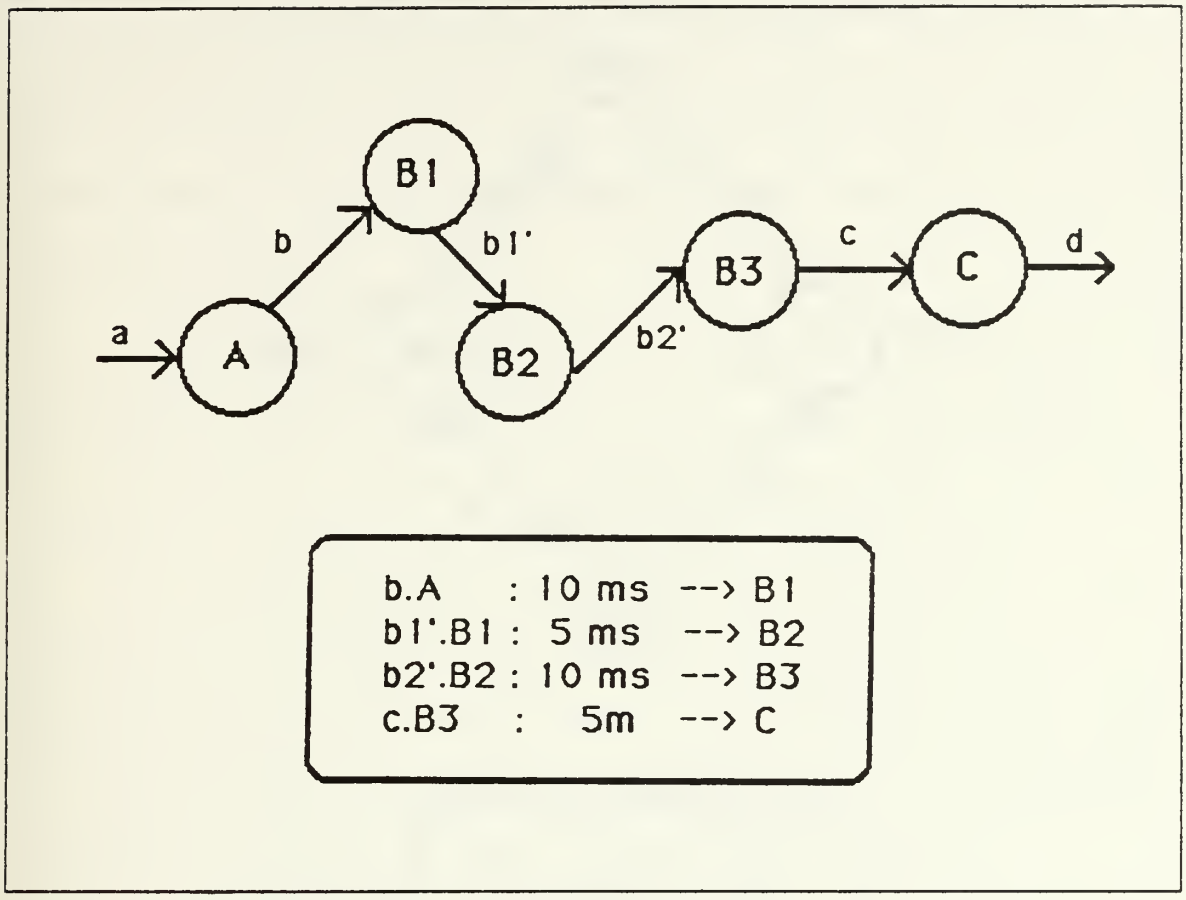


Figure 12. Decomposition of Operator B

statement having a right-hand operator with no matching left-hand operator. An augmented acyclic digraph is illustrated in Figure 13 on page 36. In this type of digraph, a decision to choose the "a.A" link first and the "d.A" link last is arbitrary. The output from either sort is a precedence list of critical operators stipulating the exact order in which they must be executed.

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that the link statements are formatted correctly. This assumption is realistic here and especially in future designs when the UI contains a syntax-directed editor. Second, although this design assumes a linear graph, the sort procedure will accommodate both linear graphs and acyclic digraphs. The linear sort will produce one precedence list while the acyclic sort can produce two or more precedence lists.

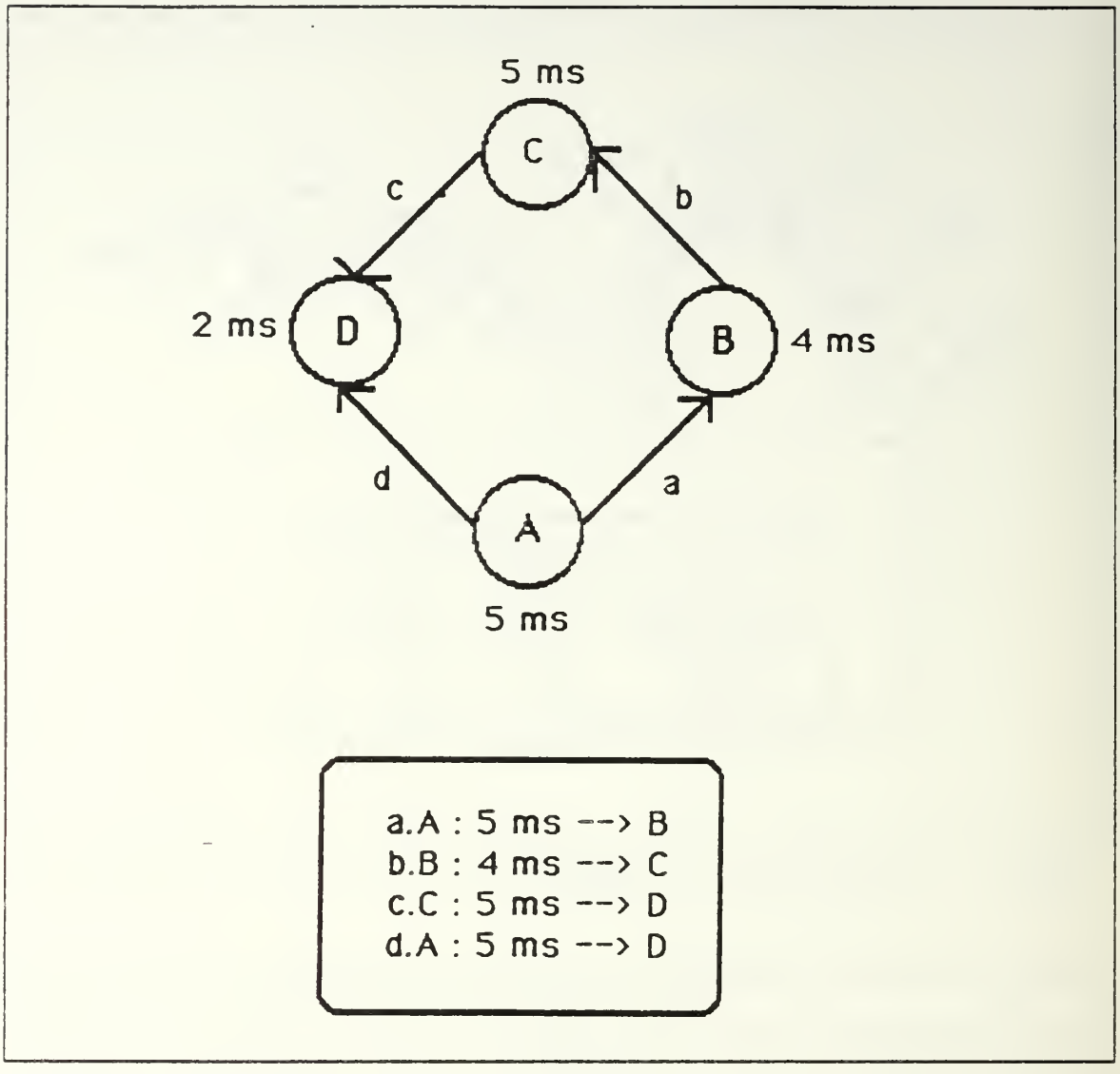


Figure 13. PSDL Augmented Acyclic Digraph

d. "Build_Harmonic_Blocks"

A second output of the "Pre-Process_File" function, the Operator file, is the input to the next major function of building harmonic blocks. An harmonic block as defined in this thesis is a set of periodic operators where the periods of all its component operators are exact multiples of a calculated base period [Ref. 4: p. 7]. This implementation design treats each harmonic block as an independent scheduling problem. When this definition is applied to scheduling hard real-time constraints, the design approach requires one processor for each harmonic block. This approach utilizes the

capabilities of concurrency and multi-processing which are normally a requirement for complex, hard real-time systems. The implementation used in this thesis addresses a single-processor environment only. Therefore, the procedures utilized in generating the final static schedule assume that only one harmonic block is created. The following sections describe how sporadic operators are converted to their periodic equivalents, how the base period is calculated and how it relates to the harmonic block, and finally, how the harmonic blocks are created.

The Operator file generated earlier contains all the critical operators with their timing constraints that were extracted from the PSDL prototype program. However, this file can initially contain both periodic and sporadic operators. Periodic operators are triggered for execution at approximately regular intervals. The resultant triggering interval, or period, is the governing factor that identifies an operator as periodic. This periodicity helps insure that execution is completed between the beginning of a period and its deadline, which defaults to the end of the period. In contrast, sporadic operators are data-driven, meaning that they are triggered by the arrival of a new data value or set of values. Attempts to create a static schedule with sporadic operators would prove difficult, if not impossible, especially when the objective of a static schedule is guaranteeing execution of operators in a predictable manner. For this reason, sporadic operators are implemented by their calculated periodic equivalent.

The first preliminary step in creating a static schedule uses an algorithm [Ref. 4: p. 8] which calculates the periodic equivalent for all sporadic operators. Use of this algorithm requires that all sporadic operators (those without periods) have values for MCP, MRT, and MET. If any of these values are missing they must be calculated from the available information. The MRT value was computed during the "Validate_Data" function. This implementation assumes that MET is given for all time critical operators and that either a period, an MCP, or both are also given for all critical operators. This author interprets the latter as indicating that an operator with both values defaults to a periodic operator. The following relationships between these values must exit to calculate a valid operator:

1. MET < MRT
2. MCP < MRT
3. MET < MCP.

The first condition insures that (MRT - MET) produces a positive value. The second condition is necessary, but it is not sufficient to insure a valid period. This condition

guarantees that an operator can fire at least once before a response is expected. The last condition insures that the period calculated will conform to a single-processor environment. [Ref. 16: p. 6] The periodic equivalent is then calculated as

$$P = \text{minimum}\,(\,MCP,\ MRT - MET\,).$$

The value of P must be greater than MET in order for the operator to complete execution within the calculated period. If this test fails, a last resort is setting P equal to MCP as a worst case, or tightest, scheduling constraint.

After all operators are in periodic form, they are sorted in ascending order based on the period values. This sort assumes that all units of time measurement were previously converted to microseconds. A second preliminary step to creating the static schedule uses an algorithm which calculates the base block and its period for the sorted sequence of operators. Within this thesis, the base period is defined as the greatest common denominator (GCD) of all operators in one sequence (or block) that will be scheduled together. Two algorithms can be used for determining the GCD. One addresses a single-processor environment only. This algorithm divides each period value in the sequence by the smallest period value. Whenever a remainder occurs, the denominator is decreased by one and the process repeats until all remainders equal zero. This algorithm results in one sequence of periods (the base block) with one base period (the GCD).

The second algorithm, applicable to multi-processor environments, is similar in design but results in one or more base blocks, each having a unique GCD and a unique sequence of operators. An initial pass through all of the periods results in two sequences, only one of which is a final base block with a GCD. When division results in a zero remainder, the period is placed in a primary sequence. When division results in a non-zero remainder, the period is placed in an alternate sequence. Subsequent passes only use the most recent alternate sequence. This process is continued until the alternate sequence equals the null set. This implementation uses the second algorithm for two reasons. First, although the basic designs are similar, the implementation is more straightforward. Second, for a single-processor environment, the second pass verifies that all periods were assigned correctly to the first sequence if the alternate sequence equals the null set.

The last preliminary step to create the static schedule uses an algorithm which calculates the length of time for the harmonic block. In a single-processor environment, the operators and their periods used to create the base block are the same as those in the harmonic block. The actual harmonic block length is the least common multiple (LCM) of all the operators' periods contained in the block. The algorithm first calculates the GCD as above for the first pair of periods in the block. The LCM is then calculated by dividing the product of this pair by the GCD. The calculated LCM is paired with the next period in the block, after which the GCD and LCM are again calculated. The LCM calculated using the last such pair is the LCM for the harmonic block. Mathematically, for a block of four periods the algorithm corresponds to

$$\text{LENGTH} = \text{LCM}\,[\,\text{LCM}\,(\,\text{Period\_3},\ \text{Period\_4}\,)\,].$$

The harmonic block and its length are an integral part of creating the static schedule. This block represents an empty timeframe within which the operators will be allocated time slots for execution.

e. "Schedule_Operators"

The "Sort_Topological" and "Build_Harmonic_Blocks" functions generated output files for Precedence_Lists and Harmonic_Blocks, respectively. Both of these files are necessary to create a static schedule for time critical operators. The Precedence_Lists file contains the required sequential execution order for all time critical operators. The Harmonic_Blocks file contains the basic timeframe within which the critical operators are allocated non-overlapping time slots. The resulting static schedule is a linear table giving the exact execution start time for each critical operator and the reserved MET within which each operator completes it execution.

The algorithm used in this implementation is a two-step process, both of which use the operators' periods and METs. The first iterative process performs two distinct functions. Initially, it allocates an execution time interval for each operator [Ref. 10: p. 126] based on

$$\text{INTERVAL} = (\ \text{current\_time},\ \text{current\_time} + \text{MET}\ ).$$

Next the process creates a firing interval for each operator during which the second iterative process must schedule the operator. The firing interval stipulates the lower and

upper bound for the next possible start time for an operator based on its period. As an example, OP_2 in Figure 14 on page 41 is scheduled to begin execution at time 2 and to complete execution by time 3 based on its MET of 1. With a period of 10, OP_2 can not fire again before time 12, the lower bound. But OP_2 must fire at or before time 21, the upper bound, in order to guarantee that execution is completed on or before time 22.

The second process has three distinct functions. Initially, it uses the lower bound of each firing interval when it schedules operators during subsequent iterations. The sequence of operators is allocated time slots according to the earliest, lower bound first. For the example in the previous figure, the operators are scheduled in the order { OP_1, OP_2, OP_3 } during the first iteration in this process. Since OP_4 has a period of 20 units and the harmonic block length is also 20 units, OP_4 is scheduled only once in each harmonic block. Before an operator is allocated a time slot, this process verifies that either:

1. (current_time + MET) < harmonic block length
2. (current_time + MET) =< harmonic block length.

The second condition is applicable to the last operator scheduled in that harmonic block only. Failure to meet either condition results in an infeasible schedule. This situation raises an EXCEPTION which halts execution since the timing constraints of that operator, or of future operators in the next iteration, will not be met.

This process also calculates new firing intervals for each operator scheduled. As an example, Figure 15 on page 42 shows the static schedule and two harmonic blocks after three iterations of this process. This example illustrates the importance of calculating an accurate harmonic block. Once all operators are correctly scheduled within an entire harmonic block, all subsequent harmonic blocks are copies of the first -- a static schedule.

2. Static Scheduler Implementation

This Chapter described the conceptualized design utilized to implement the Static Scheduler. The primary objective was providing background knowledge of the overall design using the 1st level DFD. Additional details were included to enhance the reader's understanding of the implementation guidelines presented in Chapter IV using the Ada® programming language and the lower level DFDs in Appendix C.

Given the following information:

PRECEDENCE_LISTS { OP_1, OP_2, OP_3, OP_4 }
HARMONIC_BLOCK_LENGTH = 20

OPERATOR_ID	MET	PERIOD
OP_1	2	10
OP_2	1	10
OP_3	3	20
OP_4	1	10

STATIC SCHEDULE:

OPERATOR_ID	START TIME	END TIME	FIRING INTERVAL
OP_1	0	2	(10,18)
OP_2	2	3	(12,21)
OP_3	3	6	(23,40)
OP_4	6	7	(16,25)

HARMONIC BLOCK:

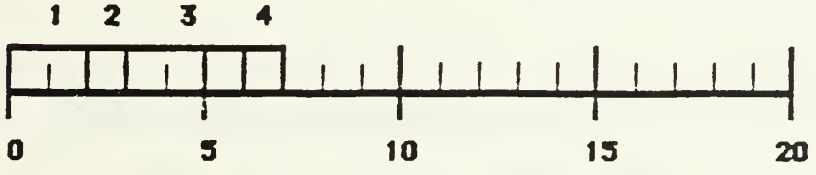


Figure 14. Static Schedule after First Process

STATIC SCHEDULE:

OPERATOR_ID	START TIME	END TIME	FIRING INTERVAL
OP_1	0	2	(10,18)
OP_2	2	3	(12,21)
OP_3	3	6	(23,40)
OP_4	6	7	(16,25)
OP_1	10	12	(20,28)
OP_2	12	13	(22,31)
OP_4	16	17	(26,35)
OP_1	20	22	(30,38)
OP_2	22	23	(32,41)
OP_3	23	26	(43,60)
OP_4	26	27	(36,45)
OP_1	30	32	(40,48)
OP_2	32	33	(42,51)
OP_4	36	37	(46,55)

HARMONIC BLOCK:

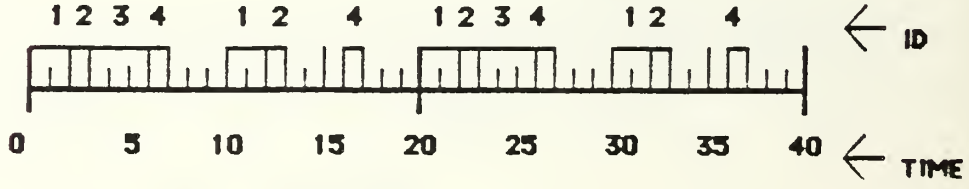


Figure 15. Static Schedule for 2 Harmonic Blocks

IV. STATIC SCHEDULER IMPLEMENTATION

The information outlined in Chapter III described the importance of the Static Scheduler to the computer-aided rapid prototyping environment and laid the groundwork for designing the implementation guidelines. Utilizing the algorithms described verbally in Chapter III and the DFDs from Appendix C, this Chapter describes the analysis, alternatives and decisions required to produce the Ada® pseudocode in Appendix E.

A. OVERALL PROGRAM STRUCTURE

As graphically shown in the 1st Level DFD in Appendix C (Figure 18 on page 67), the Static Scheduler was conceptualized as five primary functional programming units or bubbles. Each of the five bubbles was decomposed into one or more subsections which perform individual, specific functions. Together, these subsections, or lower levels, achieve the stated goal of the primary bubble. These subsections are graphically described in increasingly more detail in the 2nd, 3rd and 4th level DFDs also found in Appendix C.

Two additional primary programming units were identified during the initial analysis period. These include one programming unit comprised of all common files and a second, the main driver program. Initially the 1st Level DFD described the data flow between the bubbles, while further analysis indicated that all the bubbles utilized attributes from four primary and permanent files. Combining all four of these files in one programming unit, an Ada® library package, allows global access to all data attributes by any other bubble. Second, when using a structured programming approach, common practice requires a main driver program which provides sequential control of program execution. The driver program, implemented as a procedure in this application, contains a sequence of call statements which temporarily pass execution control to a particular programming unit. Upon execution completion by the called unit, program control returns to the driver program at the point immediately below the previously called statement. In this way, the driver program provides a thread of control through all of the program modules.

1. Naming Conventions

Before relating a specific lower level DFD to its implementation in the Ada® pseudocode, the reader requires one caveat. The 1st and 2nd level DFDs were adapted

for this thesis from a separate research effort while the 3rd and 4th level DFDs were designed during the initial stage of this research. Therefore, the names used to identify bubbles in the DFDs will not always precisely match the names associated with their pseudocode implementation. Sufficient similarity does remain which should preclude unnecessary confusion. In addition, there are cases where a lower level bubble has no unique program unit. These situations occurred when the author determined that the function described by the bubble represented a single operation and did not warrant a stand-alone program unit.

2. Implementation Approach

The remainder of this Chapter will outline the transition from a conceptualized idea to the pseudocode implementation for the Static Scheduler. Each presentation that follows begins with an analysis of the program module and the end result that must be realized. Second, any alternative approaches that will achieve the same, or similar, acceptable result are identified. Finally, if alternative solutions were identified, the presentation concludes with a description of the alternative chosen for this implementation guideline.

B. PROGRAM EXCEPTION HANDLING

Run-time errors generally occur during program execution due to hardware failure, software inconsistencies or erroneous interactive data input. An error is more precisely defined as an exception to the normal or anticipated behavior of a system. With production software, these exception situations indicate "bugs" which require either immediate or eventual modification depending on their impact to the system overall. With computer-aided rapid prototyping, however, an exception situation should alert the software designer and the user to inconsistencies in the design or parameters used to develop the prototype. In hard real-time or embedded systems, exception handling increases the reliability of critical program segments and can aid in graceful degradation of the system should exceptions occur.

1. Ada Exception Handling

The Ada® programming language includes both pre-defined and user-defined exceptions. The sets of pre-defined exceptions address conventional errors usually handled by the underlying operating system, such as DEVICE_ERROR. In contrast, user-defined exceptions can address specific input parameters or calculated values that are outside the expected or anticipated system constraints.

A standard rule for designing exception handlers stresses that exceptions should be identified and resolved at the lowest possible level in the program [Ref. 1: p. 316]. Within Ada®, after an exception is resolved, execution control returns to the end of the particular programming unit where the exception was handled. Thus, if an exception is handled within a local procedure (subroutine), the driver program is unaware of its occurrence. In a computer-aided prototyping environment, this also implies that the designer and user lose "real-time" ability to evaluate the impact of the exception.

2. Static Scheduler Exception Handling

An alternative solution, utilized for this implementation, raises the exception in the local program unit but passes exception handling to the driver program. The exceptions raised in the Static Scheduler, as shown in Figure 16 on page 46, concern the validity of the timing constraints identified by the user. Exceptions 1 through 6 indicate that either required constraints are missing or they are logically inconsistent. As an example, the third exception would be raised when MET > MRT, which indicates a negative period of time. Exceptions 7, 8 and 9 indicate that, although a schedule may be possible, there is no guarantee that it will execute within the required timing constraints. Exceptions 10, 11 and 12 indicate that, with the given timing constraints, no feasible schedule exits which meets the requirements of the envisioned system.

Depending on the system application and the user, exception handling could include built-in contingencies to modify the constraints rather than suspend execution of the prototype. However, this approach would increase the complexity of the prototype and the performance statistics reported to the designer and user.

C. PACKAGE PRESENTATIONS

The Static Scheduler, as implemented in this thesis, contains six package programming units. Five packages represent the five primary functional groupings with one additional package containing only permanent or global file information. A package is a collection of logically related, computational resources which support a structured, modular design and abstract data or type representations. Thus, the Ada® package construct provides the required mixture of accessibility and information hiding for this application. By design, the package declaration section contains sufficient visible information which facilities information exchange between programming units. At the same time, a package declaration can contain invisible (private) sections which contain structural level details that are irrelevant to the outside users. [Ref. 1: pp. 218-219] However, in this application, private types were not utilized in order that the reader could

	PACKAGE NAME	EXCEPTION NAME
1.	FILE_PROCESSOR	MET_EQUALS_PERIOD
2.	FILE_PROCESSOR	MET_NOT_LESS_THAN_MRT
3.	TOPOLOGICAL_SORTER	NO_INITIAL_LINK_OP
4.	TOPOLOGICAL_SORTER	NO_MATCHES_FOUND
5.	HARMONIC_BLOCK_BUILDER	CONSTRAINTS_INVALID
6.	HARMONIC_BLOCK_BUILDER	NO_BASE_BLOCK
7.	OPERATOR_SCHEDULER	FAIL_HALF_PERIOD
8.	OPERATOR_SCHEDULER	BAD_TOTAL_TIME
9.	OPERATOR_SCHEDULER	RATIO_TOO_BIG
10.	OPERATOR_SCHEDULER	OVER_TIME
11.	OPERATOR_SCHEDULER	INVALID_SCHEDULE
12.	OPERATOR_SCHEDULER	SCHEDULE_ERROR

Figure 16. Static Scheduler Exceptions

more easily follow program development. The following sections describe each of these packages and outline specific implementation considerations.

1. "Files" Package

The Files package groups together the four permanent files containing critical operators and their timing constraints. This structure provides a global database for use by all of the other packages as required. The Links and the Operators files are created from the information identified and retrieved from the PSDL prototype source program during the Separate_Data procedure within the File_Processor package. The Precedence_List is created by a precedence level sort routine during the Create_Lists and Sort_Remaining_Operators procedures within the Topological_Sorter package. The Schedule_Inputs file is created by two function programming units, Calc_Lower and Calc_Upper. These two sub-units are called by the Create_Interval procedure within the Operator_Scheduler package. The Files package represents an Ada® library package

which is accessible to all other programming units by specifying the "with Files" statement prior to the package declaration.

Several alternative approaches were considered for structuring this database of timing constraints. One-dimensional arrays initially appeared most appropriate with their built-in indexing structure. However, the array construct requires a priori knowledge of the array size before compilation time. The second approach considered, linked lists, would eliminate the need to include repetitive entries between files. In this application, the operator identifier must be included in each file. But this approach would also conceal the unique character or use for each file. The author believed that the record format utilized in this implementation was the most straightforward description of the individual files and their attributes.

2. "PSDL_Reader" Package

The package PSDL_Reader implements a 2nd level DFD from Appendix C (Figure 19 on page 68). Implementation assumptions for this application were outlined in Chapter III, Section B.1.a. This package represents the single most important aspect of the Static Scheduler. The procedure Invoke_AG_Processor initiates the Kodiyak AG processor that was developed in conjunction with this Static Scheduler implementation. The procedure Read_the_File uses the output of the Kodiyak AG processor as the source for all of the operators and the timing constraints needed to create the static schedule.

a. Kodiyak AG Processor

Appendix D contains a complete program listing of the Kodiyak AG processor as it was tailored for this application. The attribute grammar section of the processor contains the grammar symbols, along with their syntax and attribute definition equations, which identify the symbols within the PSDL prototype program that the Static Scheduler requires. These required symbols are the operator identifiers and their critical timing constraints (operator, maximum execution time, maximum response time, minimum calling period, time, unit, links, period, and finish).

Theoretically, the Kodiyak AG translator generator is an easy-to-use, efficient tool for creating specialized processors. However, experience showed that, in order to identify the nine required symbols, at least 30 attribute definition equations required modification. The author determined that the difficulties encountered with creating the AG processor centered around two significant areas. First, the Kodiyak AG processor is based on a tree architecture which "grows" or develops as the grammar symbols are

interpretted. Problems could possibly arise when only a small subset of the symbols are required, leaving discontinuities within the branching of the tree. Second, proficiency in using the Kodiyak AG software tool requires an extensive learning curve. Minimal documentation requires verbal "pass down" of lessons learned or repetitive trial and error to eliminate reduction errors. In addition, error reporting at compilation time is not sufficient for a student new to the area of language translators.

3. **"File_Processor" Package**

The package File_Processor implements a 2nd level DFD from Appendix C (Figure 20 on page 69). Implementation assumptions for this application were outlined in Chapter III, Section B.1.b. This package contains a sort routine and a data input validation phase. The Separate_Data (sort) procedure uses an input file, AG_Outfile, created by the PSDL_Reader package. This procedure creates three files. One of the files, Non-Crits, is used by the Static Scheduler to schedule non-time critical operators and contains operator identifiers retrieved from the AG output that do not include an MET value, a requirement for critical operators. The two files created for the Static Scheduler are the Links and the Operators files. The components of the Links file are identified within the AG_Outfile by the keywords "graph" and "links". The components of the Operators file are identified by the keywords "operators", "operator_spec", "max_exec_time", "max_resp_time", "min_period" "time", "unit", "period", and "finish".

A fourth set of retrieved symbols are dumped to a temporary file, Trash_File. These symbols arise due to a data type to operator interconnection inherent in the PSDL language. The cross connection stems from the "type_impl" (data type implementation) and the "type_spec" (data type specification) constructs in PSDL. The PSDL and AG definition equations for these symbols include operators that are utilized in the data type implementations. The author did not associate these equations with the critical operator specifications and implementations required by the Static Scheduler.

The Validate_Data procedure uses attributes from the Operators file to perform basic validity checks on the timing constraints. As described in Chapter III, certain relationships are required between timing constraints to insure, with some degree of reliability, that a static schedule is feasible. Therefore, timing constraint relationships are validated as early as possible in this implementation. An exception is raised when any constraint fails a validity test. This prompt feedback allows the designer and the user to modify constraints, when necessary, early in the prototype construction and demonstration.

4. "Topological_Sorter" Package

The package Topological_Sorter implements three lower level DFDs which appear in Appendix C (Figure 21 on page 69 through Figure 23 on page 70). Implementation assumptions for this application were outlined in Chapter III, Section B.1.c. This package utilizes two procedures and the Links file to initiate and build the Precedence_List file.

The initial procedure Create_Lists must identify the starting point for the topological sort. The procedure uses a pair of nested loops to compare the first operator identifier[i] on the left of the arrow with each and every operator identifier[j] on the right of the arrow. The operator identifier[i] with no matching identifier[j] is the starting point. Execution control passes to the next procedure after the identifiers [i] and [j] are positioned in the Precedence_List file.

The remaining link statements are sorted within the Sort_Remaining_Operators procedure. This topological sort is similar to the operation in the previous procedure, but reverses the logic. In this sort, precedence relationships among operators are evaluated by comparing the first operator[j] on the right of the arrow with the second operator[i + 1] on the left of the arrow. When a match is found, operator[i + 1] and its corresponding operator[j + 1] are placed in the second positions of the Precedence_List. This operation continues until no match is found, indicating the end of the Links file. The resulting Precedence_List is used as input to the Operator_Scheduler package.

5. "Build_Harmonic_Blocks" Package

The package Build_Harmonic_Blocks implements ten lower level DFDs from Appendix C (Figure 24 on page 71 through Figure 33 on page 75). Implementation assumptions for this application were outlined in Chapter III, Section B.1.d. This package contains four major procedures which create an harmonic block template that is tailored to the critical operators and their firing intervals.

The initial procedure Calc_Periodic_Equivalents contains three sub-procedures which perform specialized operations. The procedure Locate_Sporadic_Operators first identifies critical operators that fire sporadically, indicated by the presence of an MCP value for that operator. Second, this procedure verifies that the operator record contains an MRT value. If not present, the procedure creates the value from the given constraints as either (MRT := WITHIN) or as (MRT := MET). The second major procedure Verify_1 verifies the relationships between the timing constraint values that are needed to calculate the equivalent operator period. These constraints include the

MCP, MRT and MET values for each sporadic operator. Exceptions are raised and execution is suspended if the required relationships are not met. These exceptions indicate that a valid static schedule is not feasible with the given timing constraints. The last procedure Period_Algo calculates the equivalent period value for each sporadic operator. The new period equals the smaller value of either MCP or (MRT - MET).

The second major procedure Sort_by_Period performs an ascending order sort based on the operators' periods. The third major procedure Find_Base_Block uses this sorted operator sequence to verify the compatibility of the operators. In order to create a valid static schedule, the operators' periods must be multiples of a common divisor (GCD). A modulus (mod) division function embedded within nested loops calculates the GCD for the sequence of operator periods. For a single processor system, a single GCD for all of the operators' periods verifies that a valid static schedule exists for the given constraints.

The final major procedure Find_Block_Length calculates the actual length of the harmonic block template. This procedure uses two functions, Find_GCD and Find_LCM, that are embedded within an outer loop. This process continues through the sequence of all operator periods until a final LCM is calculated. The final LCM equals the length of the harmonic block within which the critical operators will be scheduled.

6. **"Operator_Scheduler" Package**

The package Operator_Scheduler implements ten lower level DFSs from Appendix C (Figure 34 on page 76 through Figure 43 on page 80). Implementation assumptions for this application were outlined in Chapter III, Section B.1.e. This package contains four major procedures. These procedures verify the initial feasibility of the harmonic block's accurate execution, schedule the operators within the harmonic block, and finally create the static schedule that the Dynamic Scheduler executes at prototype runtime.

The initial procedure Test_Data utilizes embedded procedures which determine the overall feasibility of the final static schedule to execute within the timing constraints. The assumptions behind these validity tests were outlined in Chapter III. Each procedure raises an exception if the static schedule inputs fail the indicated validity test. The procedure Calc_Total_Time indicates that the schedule will fail at execution since the number of operations times their METs is greater than the harmonic block length. The

procedures Calc_Half_Periods and Calc_Ratio_Sum indicate, with a high degree of probability, that the schedule will fail during execution.

The next two procedures, Schedule_Initial_Set and Schedule_Rest_of_Block, utilize the Precedence_List and the operators' MET values to create a firing interval for each operator. The firing interval contains the start time and worst case stop time, based on MET, for each operator. The stop time of operator[i], together with the period of operator[i + 1], determine the start time for operator[i + 1]. Although initially conceived as a single procedure, analysis indicated a difference in scheduling logic between the first and subsequent passes through the Precedence_List. During the second and future passes the operators' execution order may vary from the Precedence_List based on each operators' period. The final output of these two major procedures is the Schedule_Inputs file which contains a pre-allocated firing interval (slot) within the harmonic block for each time critical operator.

The final procedure Create_Static_Schedule utilizes the Schedule_Inputs file to create the static schedule used by the Dynamic Scheduler at run-time. As conceived by the author, the static schedule is implemented as an Ada® package containing a task programming unit. The task, in turn, contains the Ada® rendezvous statements ENTRY and ACCEPT. The ENTRY statements call either the next scheduled operator or the Dynamic Scheduler. The Dynamic Scheduler is called when either a non-allocated time slot is reached or when a critical operator completes execution prior to its scheduled stop time.

D. RUN-TIME STATIC SCHEDULE

The final output of the Static_Scheduler program, the Static_Schedule file, provides the input for the CLE. The CLE compiles the Static Scheduler's output and produces an executable Ada® program. An example of the compiled program is shown in Figure 17 on page 52. This program, together with the executable program from the ESS Translator, provide the executable prototype program for the envisioned system.

```
package THE_STATIC_SCHEDULE is
  task SEQUENCE_OF_CALLS is
    entry EACH_OPERATOR.OPERATOR[i] (THE_START[i] : in out INTEGER;
                                     THE_STOP[i]  : in out INTEGER);
    entry DYNAMIC (THE_STOP[i]    : in out INTEGER;
                   THE_START[i+1] : in out INTEGER);
  end SEQUENCE_OF_CALLS;
end THE_STATIC_SCHEDULE;

with CALENDAR;
package body THE_STATIC_SCHEDULE is
begin
  task body SEQUENCE_OF_CALLS is
  begin
    loop
      procedure THE_OPERATOR[i] is
      begin
       accept EACH_OPERATOR.OPERATOR[i](THE_START[i] : in out INTEGER;
                                        THE_STOP[i] : in out INTEGER);
        select
          when CLOCK.VALUE < THE_STOP[i] then
            entry DYNAMIC
        or
          when CLOCK.VALUE = THE_STOP[i] then
            entry EACH_OPERATOR ( OPERATOR[i] )
        else
          exception
            when others =>
              raise RUNTIME_MET_FAILURE
      end THE_OPERATOR;
    end loop;
  end SEQUENCE_OF_CALLS;

end THE_STATIC_SCHEDULE;
```

Figure 17. Ada® Pseudocode for the Static Schedule

V. TELECOMMUNICATIONS APPLICATION

The utilization of digital computers in telecommunications systems, in general, greatly increases the DOD's and DON's ability to meet the growing needs of users, both ashore and afloat. The advent of embedded computer systems which preform automated circuit switching, whether at a station's technical control facility or within the satellite itself, will again increase the service department's ability to expand or add new telecommunications capabilities. The growth of and demand for communications services would benefit from the development of dynamic circuit or channel assignment systems. These systems provide improved allocation of scarce satellite and radio frequency spectrum resources. Computer-aided rapid prototyping, and specifically the CAPS, will assist in the design and development of these complex telecommunications systems.

A Satellite-Switched Time Division Multiple Access (SS/TDMA) system is a specific example of a telecommunications system whose development would benefit from the use of CAPS. In this system, communications between earth stations in different geographic areas is achieved when the satellite switches (changes) transmission time slots from one beam to another. Only one station from each individually covered area can transmit to the satellite at the same time. If N areas are served, then the satellite switch is configured to handle N! modes for complete connectivity among all of the stations. A mode represents a defined uplink-downlink connection from each station to every other station. [Ref. 17: pp. 309-311] The number of possible modes quickly becomes unmanageable without automated switching systems. The complexity and precision required for designing and developing such a system can be represented by the PSDL computational model used in the CAPS.

The CAPS described in this thesis provides a development tool which is easily adapted to modeling real-time or embedded switching systems. The PSDL computational model (see Chapter II), based on data flow through a system, represents an abstract view of both the hardware and software of these complex systems. As adapted for the SS/TDMA system, the computational model represents

$$G = (V, E, T(v), C(v))$$

where:

- V = a set of operators (signal processing equipment)
- E = a set of data streams (up and downlink beams)
- T(v) = MET for each operator (v)
- C(v) = a set of control constraints for each operator (environmental factors and mode configurations).

These MET timing and control constraints are declarations about the system itself and the environment in which it must function. This is in contrast to instructions within the applications software.

The communications switching system aboard the satellite receives a signal via a channel on the uplink beam. The system then processes the signal in some manner depending on the application. An MET associated with this process insures proper execution of the TDMA function and of the system as a whole. Control constraints for this system would include, for example, the mode configuration routing tables, data error and flow control, and system load capacity. Once the switching process is complete, the system transmits the signal on the appropriate downlink beam. When the above information is applied to the CAPS and the prototype is executed, the resultant demonstrations of the prototype can be used to validate the design specifications for the receive, process and transmit functions of the envisioned SS/TDMA system.

As users of military telecommunications systems become accustomed to high-speed, reliable and dynamic communications, applications for hard real-time or embedded systems will increase. These systems, especially switching systems, are characterised by multiplicity and concurrency of otherwise simple, interacting functions. The conventional approach to software life cycle development is quickly becoming inadeaquate or obsolete for the strenuous demands of designing and developing these complex systems. Abstract modeling of telecommunications systems reduces the probability of development failure by identifying areas with the highest risk. Design efforts are then concentrated on reducing those risks by demonstrations of the prototype early in the design stages. The CAPS tool will assist procurement and development agencies in meeting the telecommunications needs of the future.

VI. CONCLUSION

The goals of this pioneering effort were to demonstrate the feasibility of implementing a Static Scheduler for the CAPS and to provide guidelines for its implementation. This thesis outlines the tools and algorithms that are required, at a minimum, to implement the Static Scheduler and to integrate it within the Execution Support System. This study accomplished these goals while identifying specific areas of concern for future research.

The Kodiyak AG translator generator is an effective and efficient tool for processing a PSDL prototype source program. An AG processor designed specifically and precisely for the Static Scheduler is fundamental to the successful scheduling of time critical operators. Misrepresentation of or failure to identify operators and their timing constraints negates the benefits of the best designed scheduling algorithms. Lack of user friendly error messages and basic manuals causes an extensive learning period using trial and error or verbal "pass down".

The Ada® programming language provides the necessary constructs and enforces a modularized, self-documenting design, both of which enhance the feasibility of implementing the Static Scheduler. Ada® user-defined file and data types allow precise declaration (definition) of critical operators' attributes (timing constraints). Ada® rendezvous operations using ENTRY and ACCEPT statements provide a means to establish and enforce execution precedence among critical operators. Rendezvous operations provide the backbone of the runtime static schedule created by the Static Scheduler. Formal demonstration of the Static Scheduler will determine whether these Ada® constructs are sufficient and effective in meeting the critical timing constraints of hard real-time or embedded systems.

Recognizing the increased cost and importance of software development for Command and Control systems, the Secretary of the Navy (SECNAV) promulgated a new instruction addressing software development and acquisition (see Ref. 18). This instruction documents SECNAV concern for defining a DON acquisition policy for software-intensive systems and increasing user involvement during the design and development stages. The policy combining these two concerns states

> To promote effective interaction between the user and the developer, software prototyping methods shall be used in the design and construction of C2 information systems. Early delivery of software systems is emphasized through the use of prototyping methods. [Ref. 18: p. 2]

The instruction defines software prototypes identical to that used throughout this thesis as

> Software which stimulates the important interfaces and performs the main functions of the intended system while not being bound by the same hardware, speed, size or cost constraints. It may serve to demonstrate or provide a subset of the functions that would be required of software to meet a related, fully-validated requirement. [Ref. 18: pp. 1-2]

Computer-aided rapid prototyping specifically addresses the concerns of the SECNAV. In particular, the CAPS stresses interaction between the software designer and the user early in the design and development stages. This allows validation of the prototype's ability to simulate the critical interfaces and functions of the envisioned system. The author agrees that the increased cost and complexity of developing software warrants a revised approach to the software acquisition procedures.

Concurrent research projects to conceptualize components of the CAPS Execution Support System are now complete. These individual efforts empirically demonstrate that the initial goal of providing an automated execution environment for a software design or specification prototypes is feasible. The CAPS will provide software designers with an automated tool which allows validation of prototypes for hard real-time or embedded systems before extensive time and money are invested in production software. Additional research projects are currently underway to conceptualize, implement and integrate the various components or subsystems of CAPS. The time and effort expended today toward a formal demonstration of the CAPS, together with increased usage of the Ada® programming language, promise a future rapid prototyping environment which meets the demanding needs of the DOD and DON software procurement and development process.

APPENDIX A. PSDL GRAMMAR

Optional items are enclosed in [square brackets] and items that may appear zero or more times appear in { braces }
Terminal symbols appear in " double quotes ".

```
psdl = { component }

component = data_type
          | operator

data_type = "type" id type_spec type_impl

operator = "operator" id operator_spec operator_impl

type_spec = "specification" [ type_decl ]
            { "operator" id operator_spec }
            [ functionality ] "end"

operator_spec = "specification" interface
                [ functionality ] "end"

interface = { attribute [ reqmts_trace ] }

attribute = generic_param
          | input
          | output
          | states
          | exceptions
          | timing_info

generic_param = "generic" type_decl

input = "input" type_decl

output = "output" type_decl

states = "states" type_decl "initially" expression_list

exceptions = "exception" id_list

id_list = id { "," id }

timing_info = [ "maximum execution time" time ]
              [ "minimum calling period" time ]
              [ "maximum response time" time ]

time = integer [ unit ]

unit = "microsec"
     | "ms"
     | "sec"
     | "min"
```

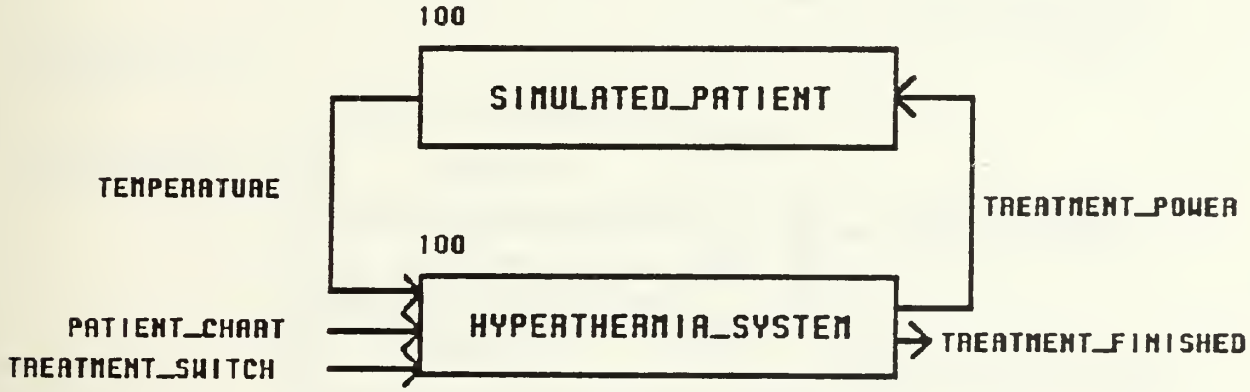
```
      | "hours"

reqmts_trace = "by requirements" id_list


functionality = [ keywords ]
                [ informal_desc ]
                [ formal_desc ]

keywords = "keywords" id_list

informal_desc = "description" " { " text " } "

formal_desc = "axioms" " { " text " } "

type_impl = "implementation" "Ada" id
          | "implementation" type_name
            { "operator" id operator_impl } "end"

operator_impl = "implementation" "Ada" id
              | "implementation" psdl_impl

psdl_impl = data_flow_diagram
            [ streams ]
            [ timers ]
            [ control_constraints ]
            [ informal_desc ]
              "end"

data_flow_diagram = "graph" { link }

link = id "." op_id "->" id

op_id = id [ ":" time ]

streams = "data stream" type_decl

type_decl = id_list ":" type_name { "," id_list ":" type_name }

type_name = id
          | id " [ " type_decl " ] "

timers = "timer" id_list

control_constraints = "control constraints" { constraint }

constraint = "operator" id
             [ "triggered" [ trigger ] [ "if" predicate ]
                           [ reqmts_trace ] ]
             [ "period" time [ reqmts_trace ] ]
             [ "finish within" time [ reqmts_trace ] ]
             { "output" id_list "if" predicate
               [ reqmts_trace ] }
             { "exception" id [ "if" predicate ]
               [ reqmts_trace ] }
             { timer_op id [ "if" predicate ]
```

```
                [ reqmts_trace ] }

timer_op = "RESET timer"
         | "START timer"
         | "STOP timer"
         | "READ timer"

trigger = "by all" id_list
        | "by some" id_list

predicate = "not" predicate
          | predicate "and" predicate
          | predicate "or" predicate
          | expression
          | id ":" id_list

expression = constant
           | id
           | type_name "." id "(" expression_list ")"

expression_list = [ expression { "," expression } ]
```

APPENDIX B. HYPERTHERMIA SYSTEM

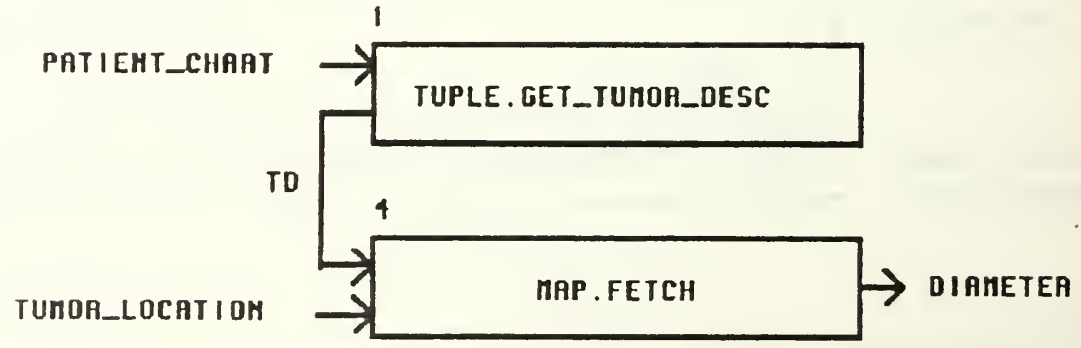
This appendix contains the PSDL source program for the Hyperthermia System as created by the designer and the user. This program does not represent all aspects of the final envisioned system. As required for prototype development with the Computer Aided Prototyping System (CAPS), this PSDL program contains only those attributes and functions that are required to simulate the critical timing characteristics of the system.

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart:   medical_history,
    treatment_switch:   boolean
 OUTPUT treatment_finished:   boolean
 STATES temperature:   real
  INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END

IMPLEMENTATION
 GRAPH
```

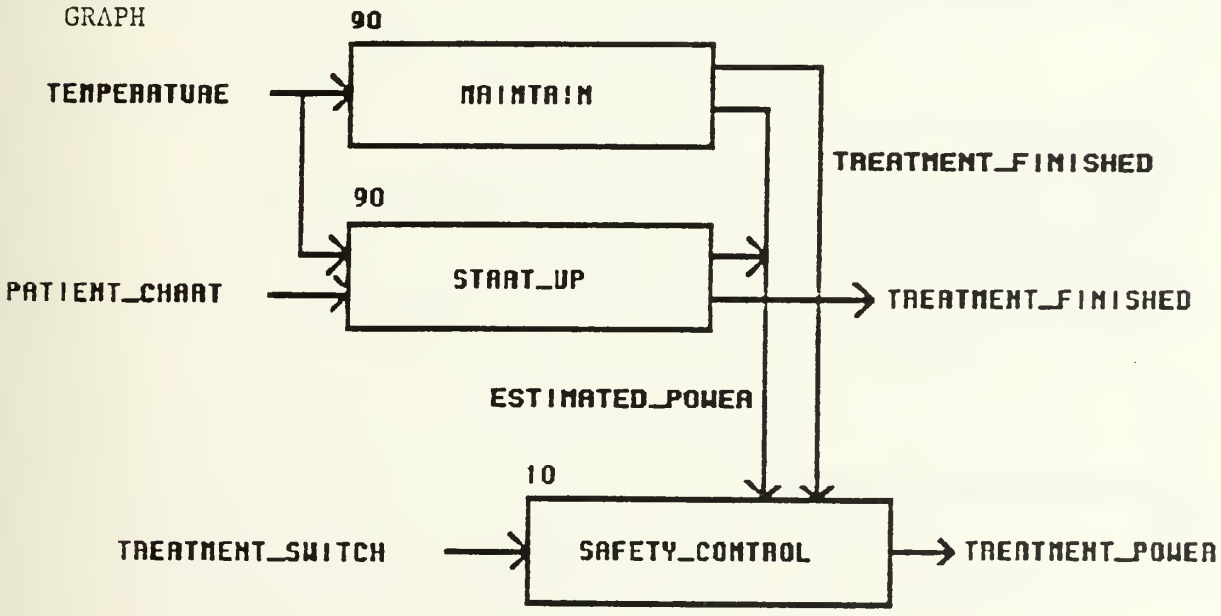



```
 DATA STREAM treatment_power:   real
 CONTROL CONSTRAINTS
  OPERATOR hyperthermia_system
   PERIOD 200 BY REQUIREMENTS shutdown
  OPERATOR simulated_patient
```

```
   PERIOD 200
 DESCRIPTION { paraphrased output }
END


TYPE medical_history
SPECIFICATION
 OPERATOR get_tumor_diameter
  SPECIFICATION
   INPUT patient_chart:  medical_history,
         tumor_location:  string
   OUTPUT diameter:  real
   EXCEPTIONS no_tumor
   MAXIMUM EXECUTION TIME 5 ms
   DESCRIPTION
   { Returns the diameter of the tumor at a given location,
     produces an exception if no tumor at that location.
   }
 END


 KEYWORDS patient_charts, medical_records, treatment_records,
    lab_records
 DESCRIPTION
 { The medical history contains all of the disease and
   treatment information for one patient.  The operations
   for adding and retrieving information not needed by
   the hyperthermia system are not shown here.
 }
END


IMPLEMENTATION
 tuple [tumor_desc:  map[from:  string, to:  real], ... ]

 OPERATOR get_tumor_diameter
  IMPLEMENTATION
   GRAPH
```



```
    DATA STREAM td:  tumor_descr
    CONTROL CONSTRAINTS
     OPERATOR map.fetch
      EXCEPTIONS no_tumor IF not(map.has(tumor_location, td))
   END


END


OPERATOR hyperthermia_system
SPECIFICATION
 INPUT temperature:  real, patient_chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 100 ms
  BY REQUIREMENTS temperature_tolerance
 MAXIMUM RESPONSE TIME 300 ms
  BY REQUIREMENTS shutdown
 KEYWORDS medical_equipment, temperature_control,
    hyperthermia, brain_tumors
 DESCRIPTION
  { After the doctor turns on the treatment switch, the
    hyperthermia system reads the patient's medical record
    and turns on the microwave generator to heat the tumor
    in the patient's brain.  The system controls the power
    level to maintain the hyperthermia temperature of
    42.5 degrees C. for 45 minutes to kill the tumor cells.
    When the treatment is over, the system turns off the
    power and notifies the doctor.
  }
 END


IMPLEMENTATION
 GRAPH
```



```
 DATA STREAM estimated_power:  real
 TIMER treatment_time
  CONTROL CONSTRAINTS
   OPERATOR start_up
    TRIGGERED IF temperature < 42.4
     BY REQUIREMENTS maximum_temperature
    STOP TIMER treatment_time
    RESET TIMER treatment_time IF temperature <= 37.0

   OPERATOR maintain
    TRIGGERED IF temperature >= 42.4
     BY REQUIREMENTS maximum_temperature
    START TIMER treatment_time
     BY REQUIREMENTS treatment_time, temperature_tolerance
    OUTPUT treatment_finished IF treatment_time >= 45 min
     BY REQUIREMENTS treatment_time
END


OPERATOR start_up
SPECIFICATION
 INPUT patient_chart:  medical_history, temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
  BY REQUIREMENTS startup_time
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { Extracts the tumor diameter from the medical history and
   uses it to calculate the maximum safe treatment power.
   Estimated-power is zero if no tumor is present.  The
   treatment finished is true only if no tumor is present.
 }
END


IMPLEMENTATION Ada start_up
END


OPERATOR maintain
SPECIFICATION
 INPUT temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { The power is controlled to keep the power between 42.4
   and 42.6 degrees C.
 }
END


IMPLEMENTATION Ada maintain
END
```

```
OPERATOR safety_control
SPECIFICATION
 INPUT treatment_switch, treatment_finished:  boolean
    estimated_power:  real
 OUTPUT treatment_power:  real
  BY REQUIREMENTS shutdown
 MAXIMUM EXECUTION TIME 10 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { The treatment power is equal to the estimated power
   if the treatment switch is true and treatment finished
   is false.  Otherwise the treatment power is zero.
 }
END


IMPLEMENTATION Ada start_up
END
```

APPENDIX C. STATIC SCHEDULER DATA FLOW DIAGRAMS

This appendix contains the Data Flow Diagrams (DFDs) for the Static Scheduler. The 1st and 2nd level DFDs from Reference 15 provide the groundwork for describing the Static Scheduler in Chapter III. The lower level diagrams represent the pioneering efforts for implementing the Static Scheduler in Chapter IV of this thesis.

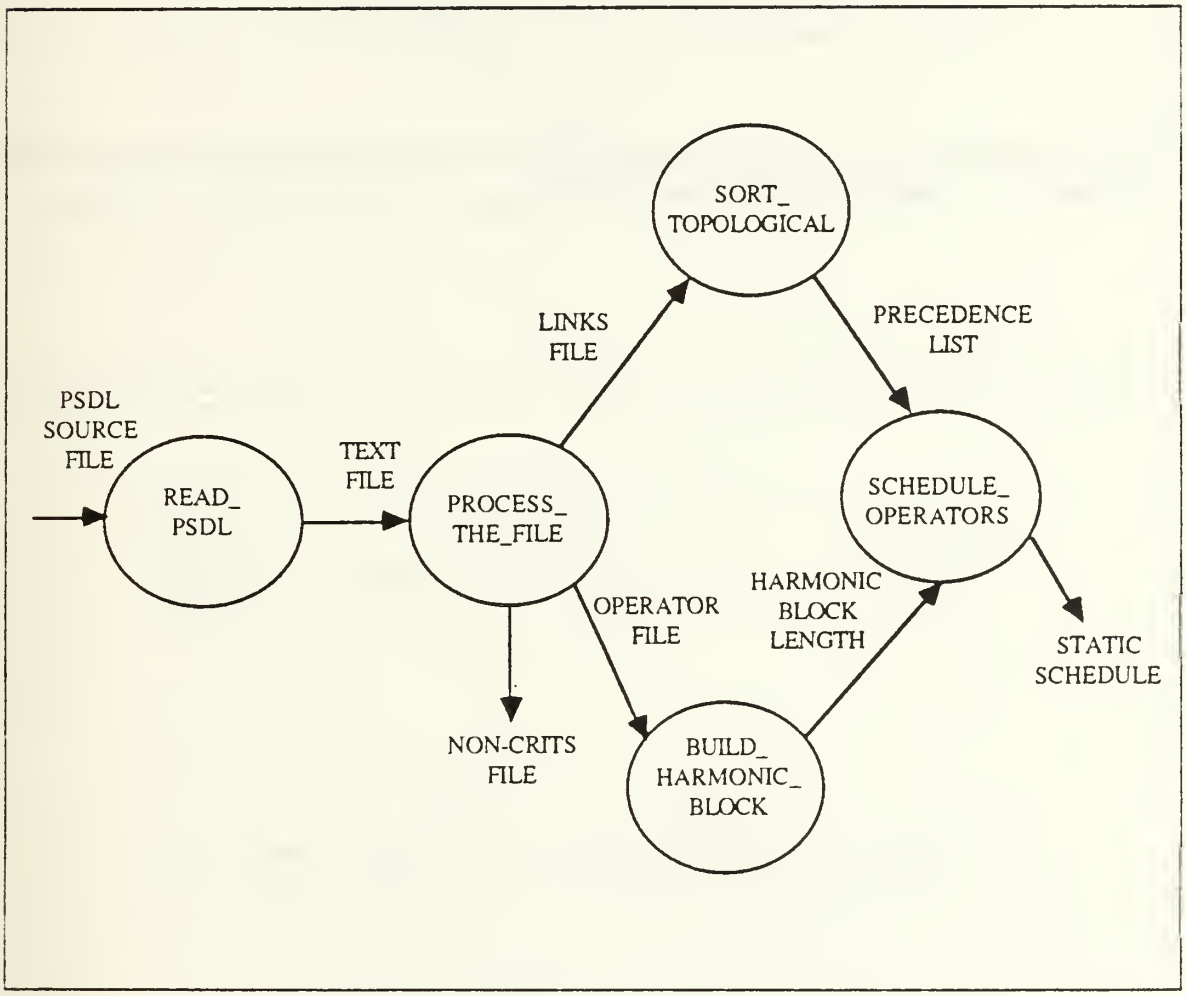


Figure 18. 1st Level DFD

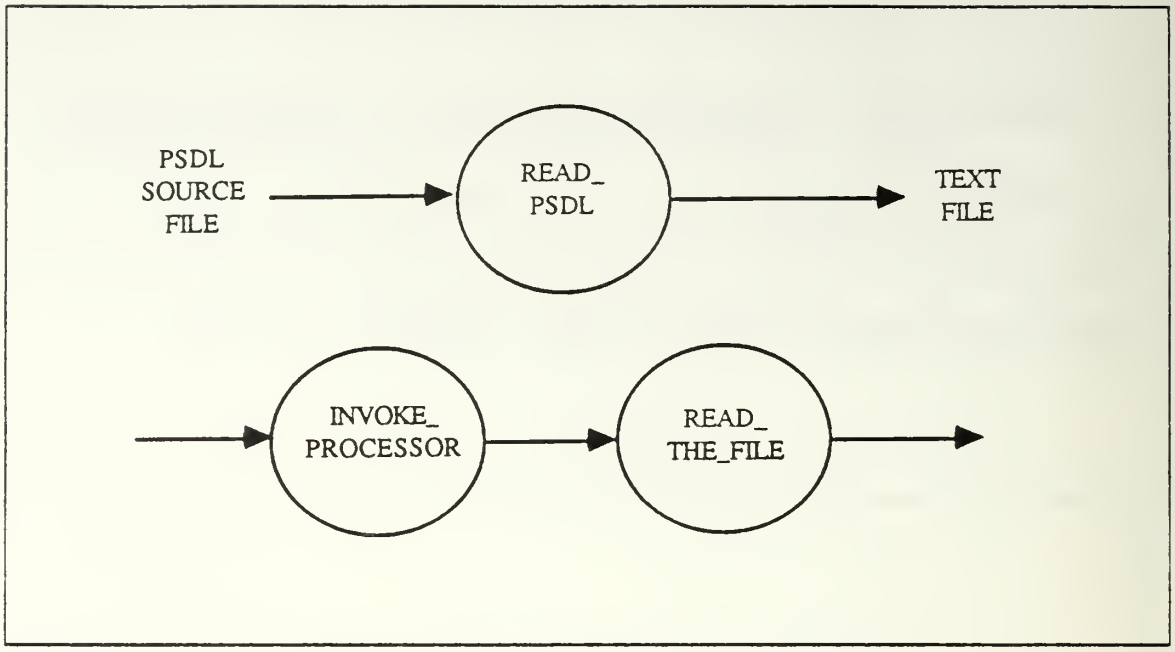


Figure 19. 2nd Level DFD "Read_PSDL"

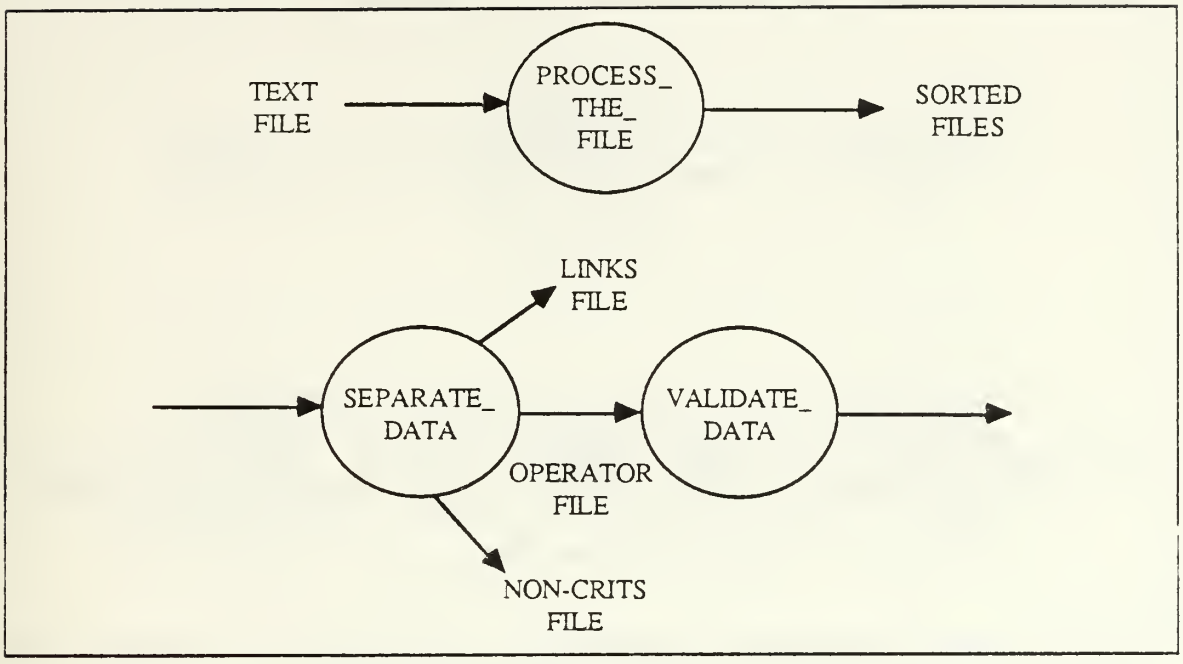


Figure 20. 2nd Level DFD "Process_the_File"

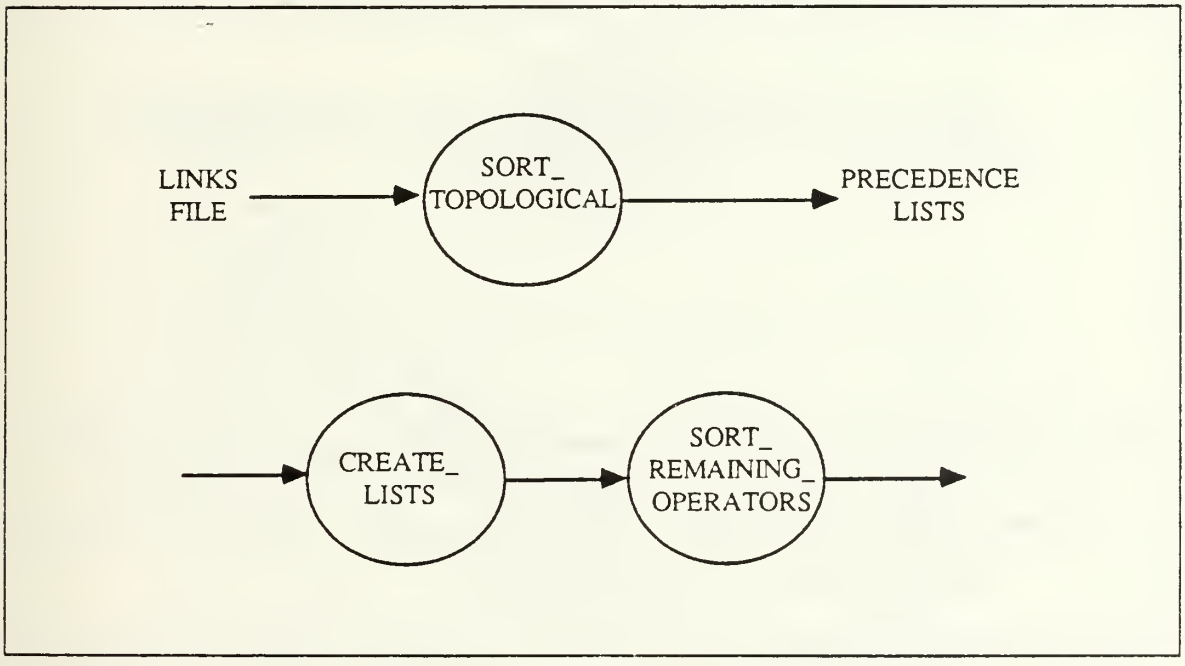


Figure 21. 2nd Level DFD "Sort_Topological"

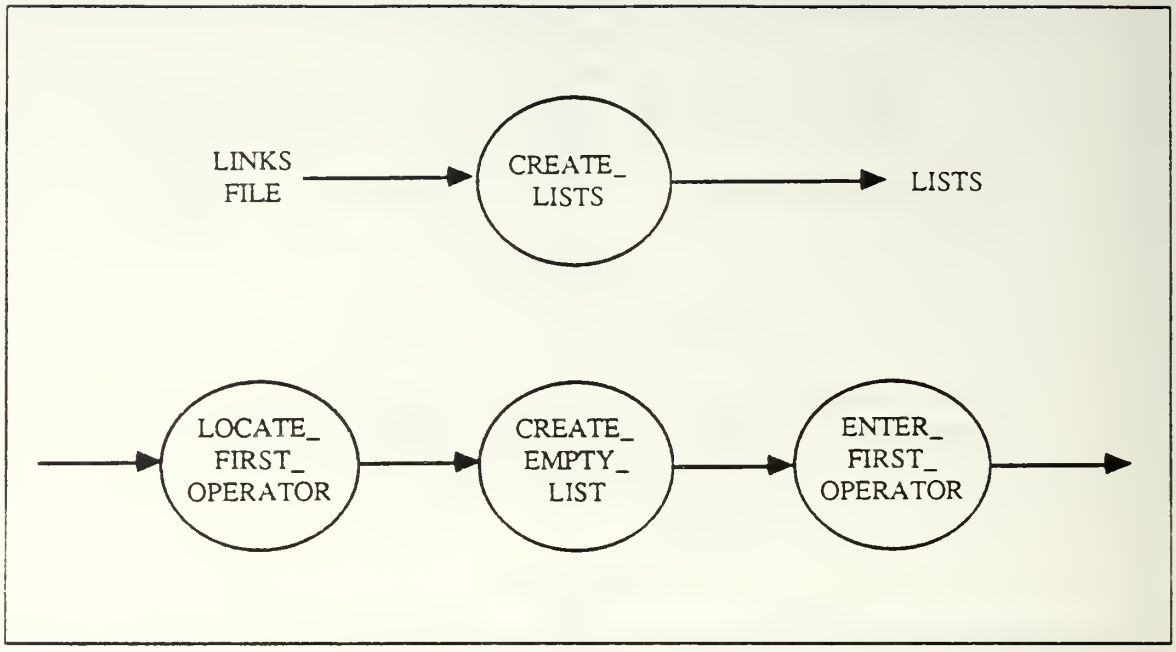


Figure 22. 3rd Level DFD "Create_Lists"

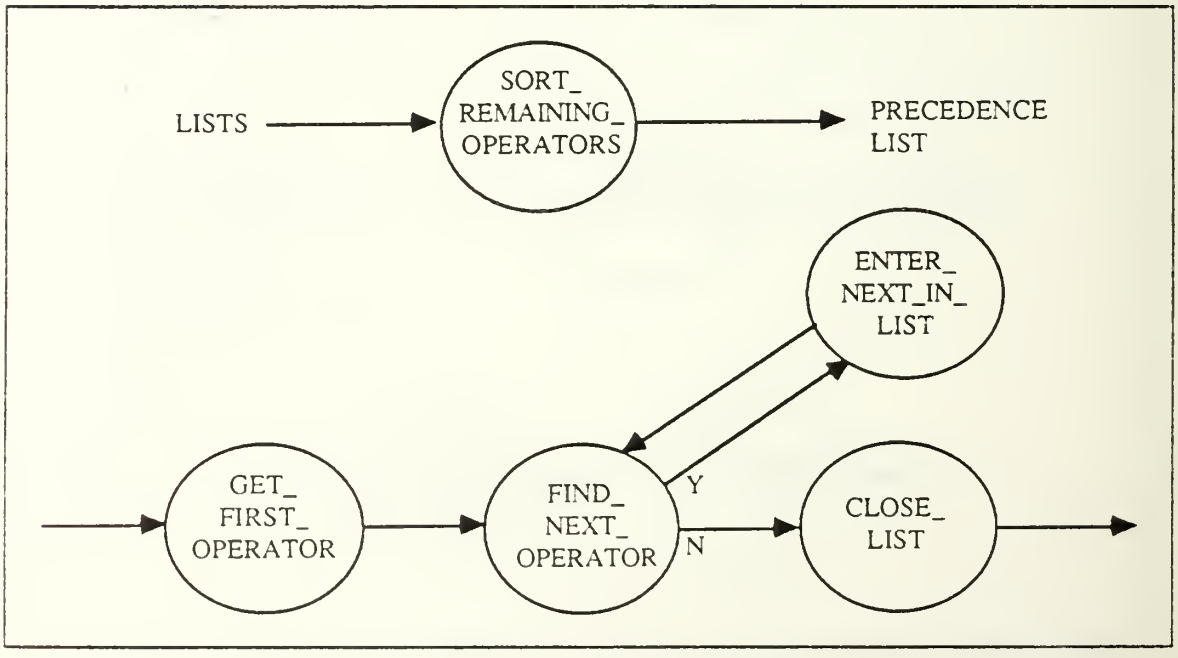


Figure 23. 3rd Level DFD "Sort_Remaining_Operators"

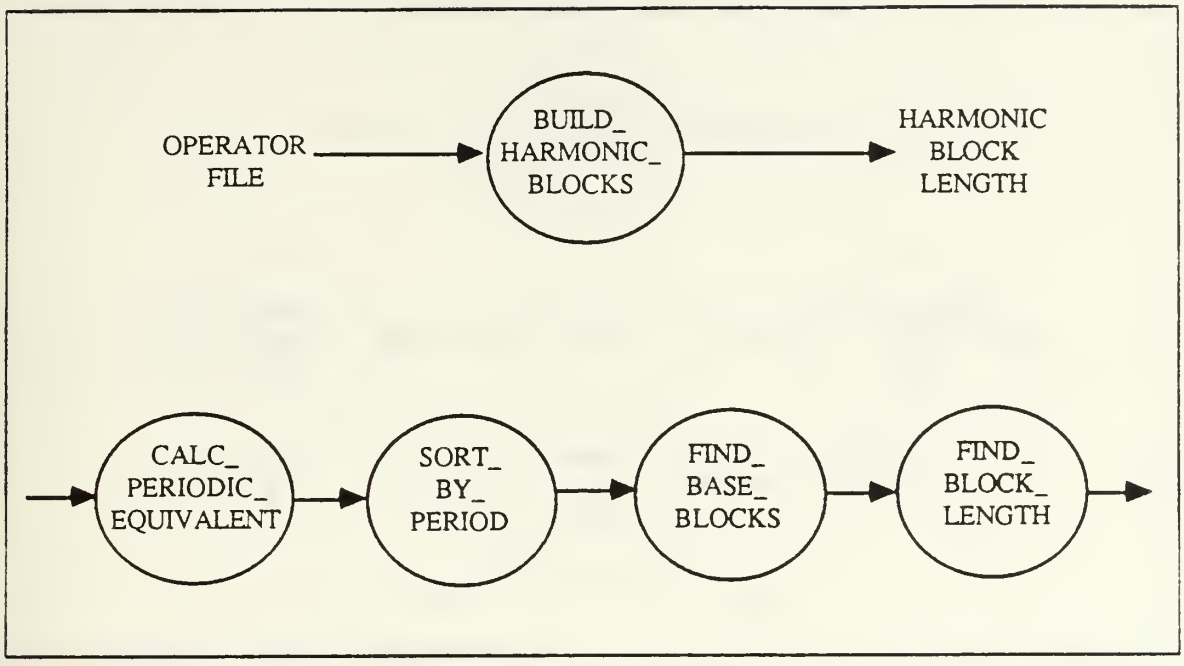


Figure 24. 2nd Level DFD "Build_Harmonic_Blocks"

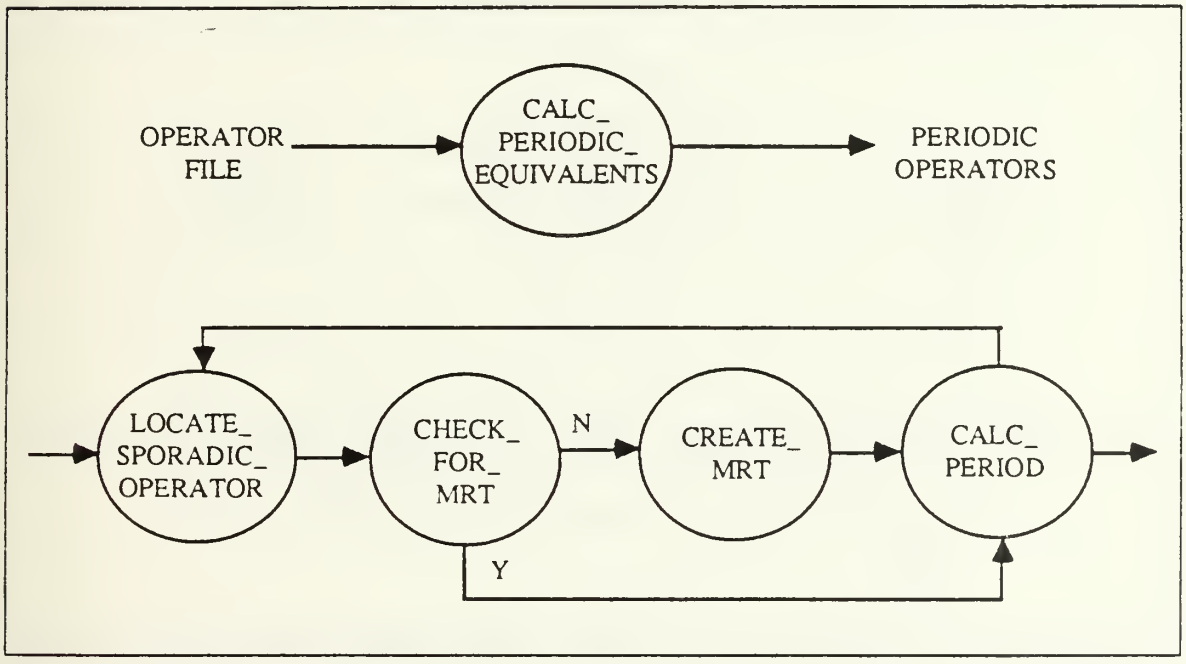


Figure 25. 3rd Level DFD "Calc_Periodic_Equivalents"

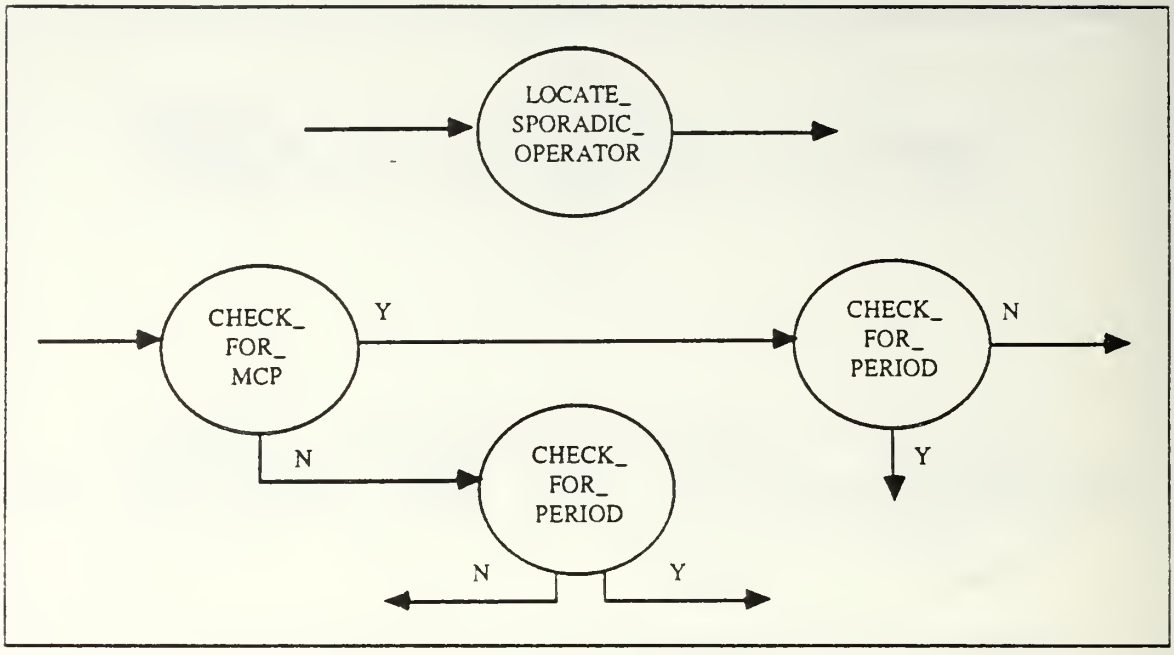


Figure 26. 4th Level DFD "Locate_Sporadic_Operator"

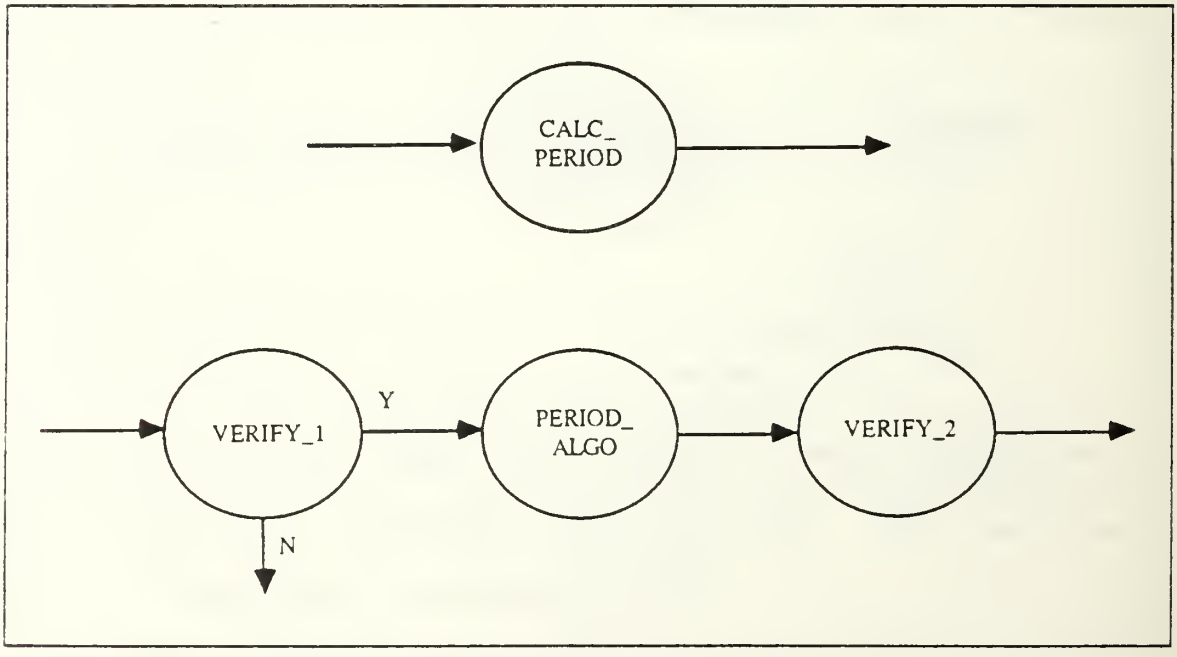


Figure 27. 4th Level DFD "Calc_Period"

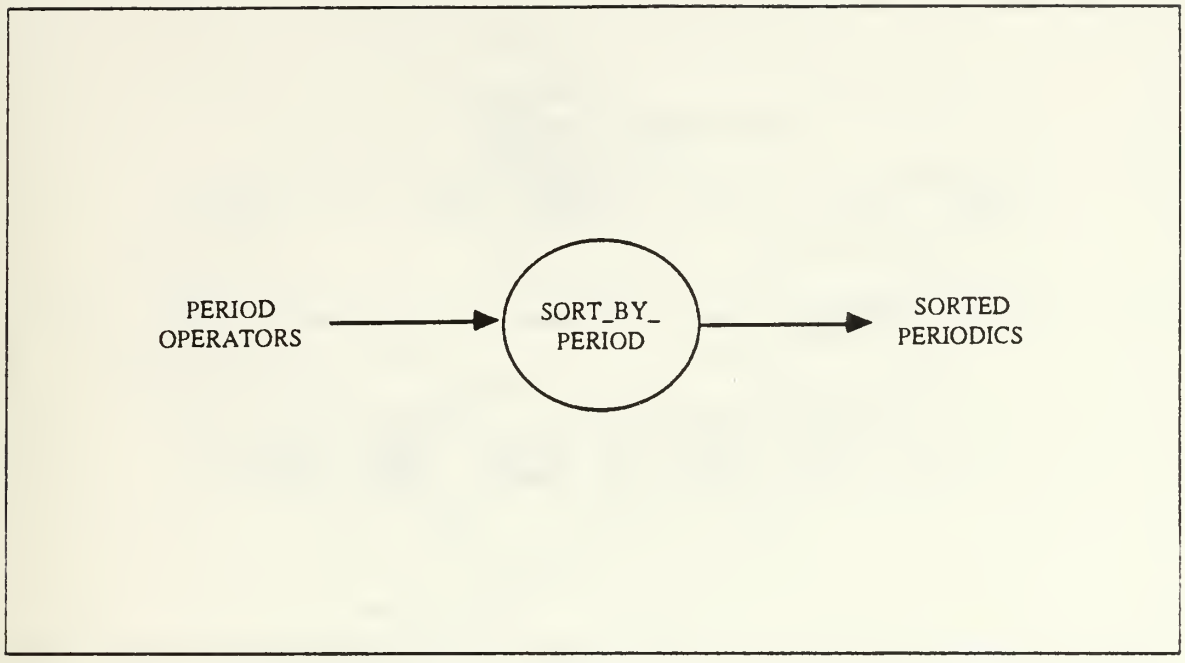


Figure 28. 3rd Level DFD "Sort_by_Period"

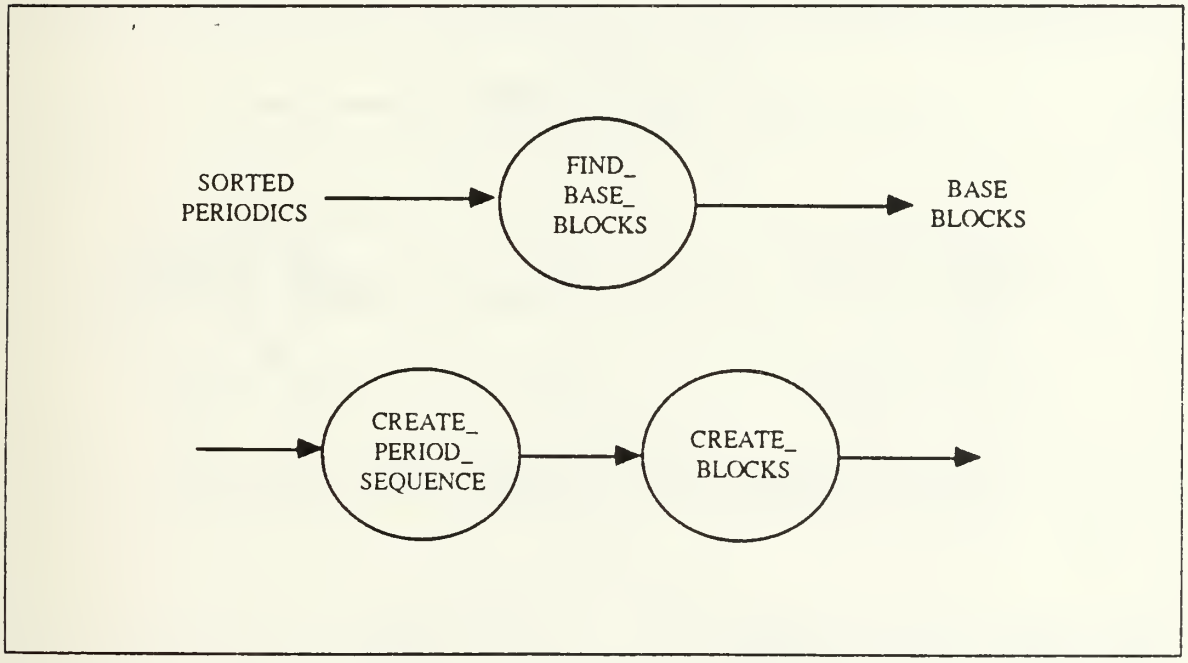


Figure 29. 3rd Level DFD "Find_Base_Blocks"

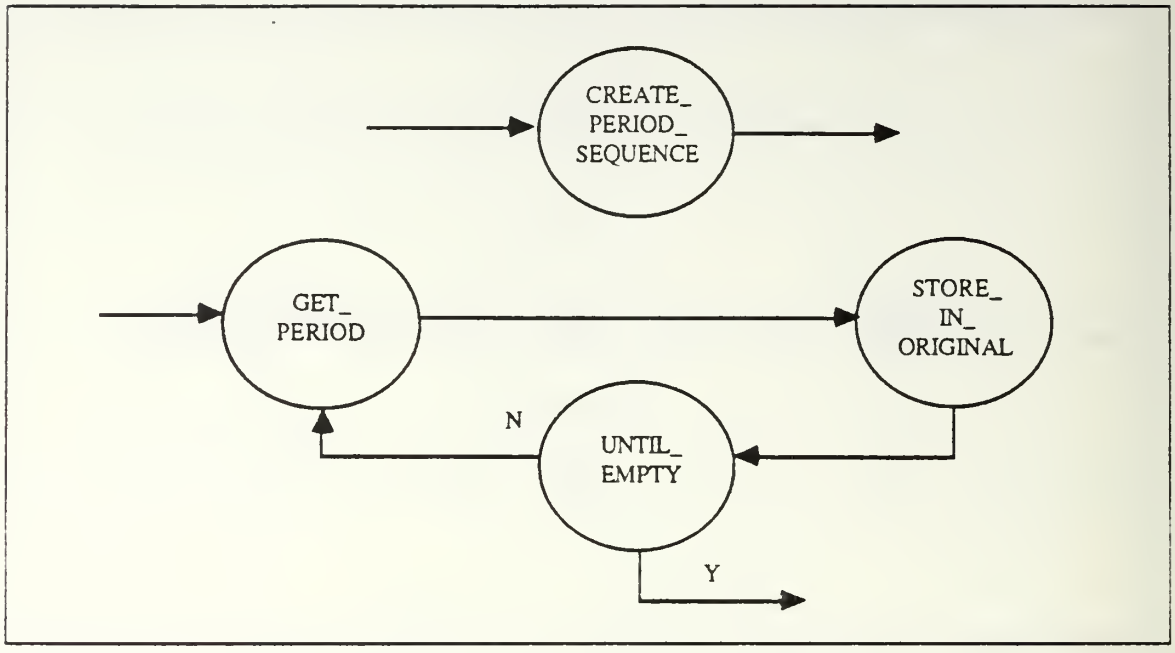


Figure 30. 4th Level DFD "Create_Period_Sequence"

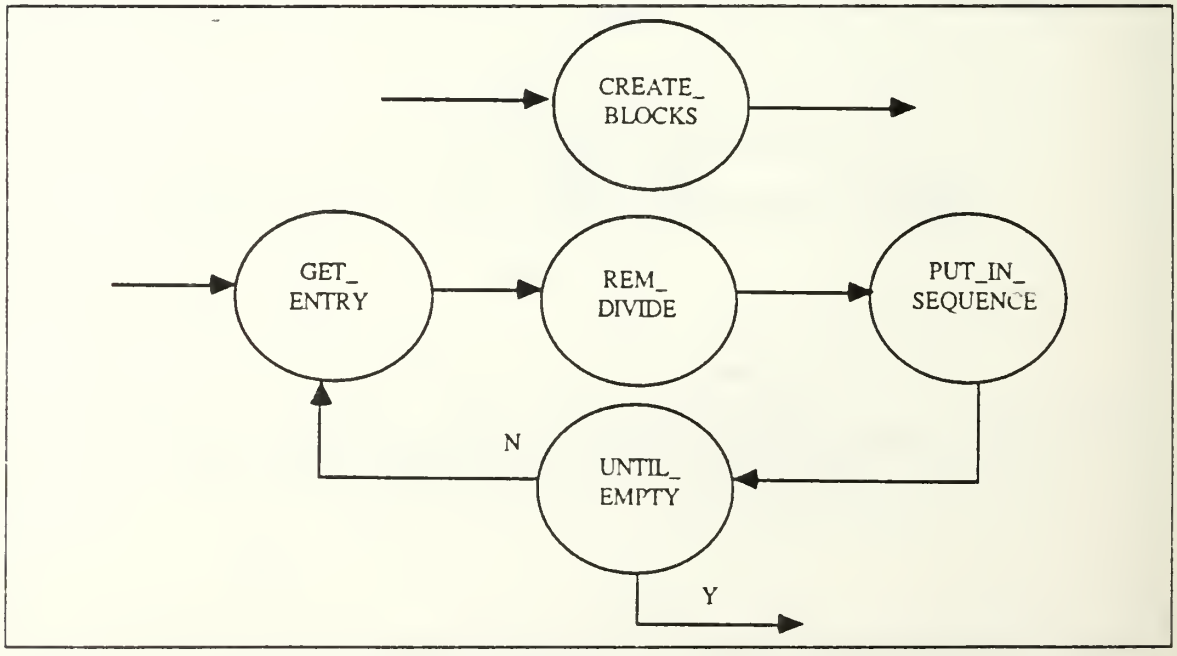


Figure 31. 4th Level DFD "Create_Blocks"

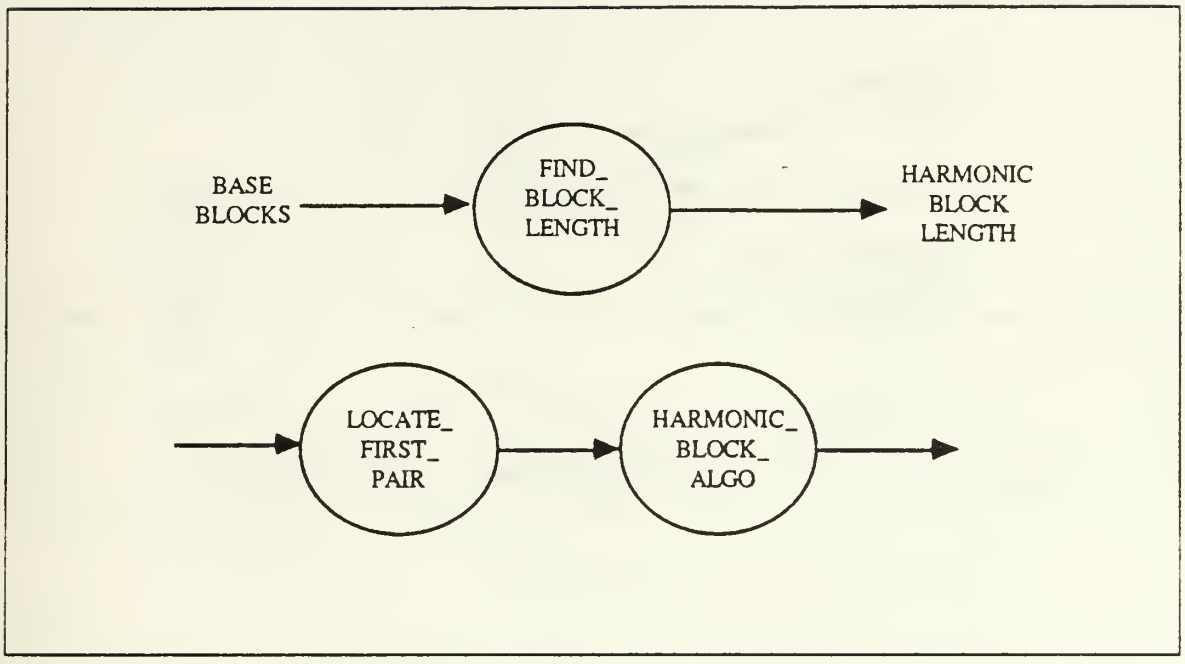


Figure 32. 3rd Level DFD "Find_Block_Length"

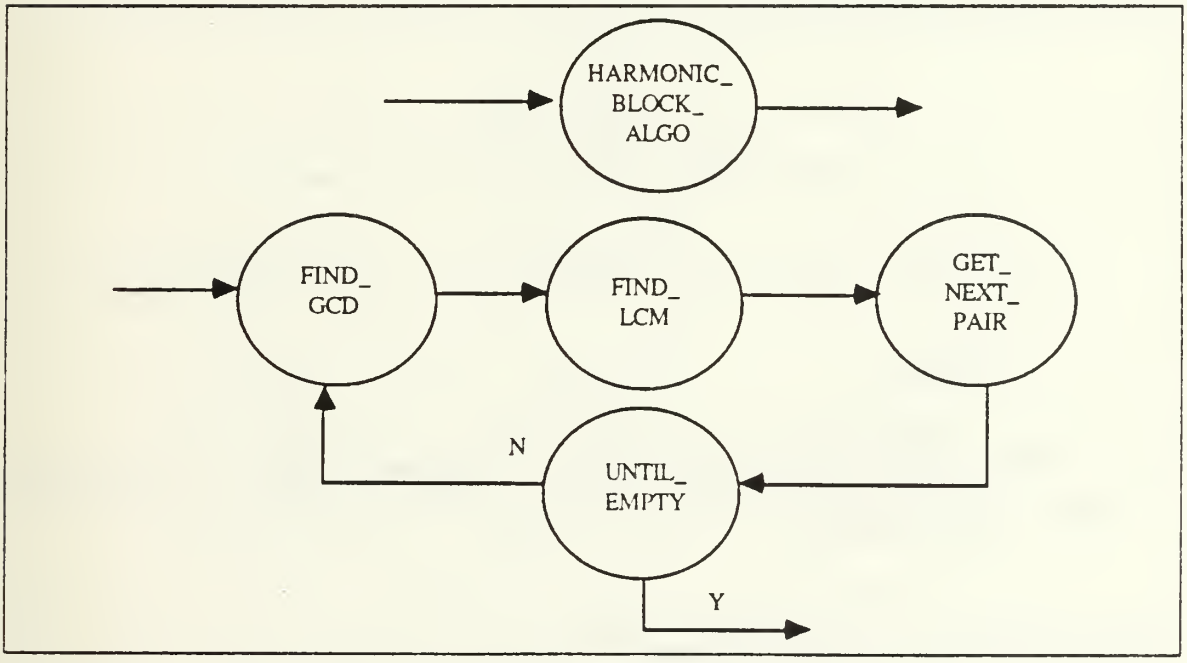


Figure 33. 4th Level DFD "Harmonic_Block_Algo"

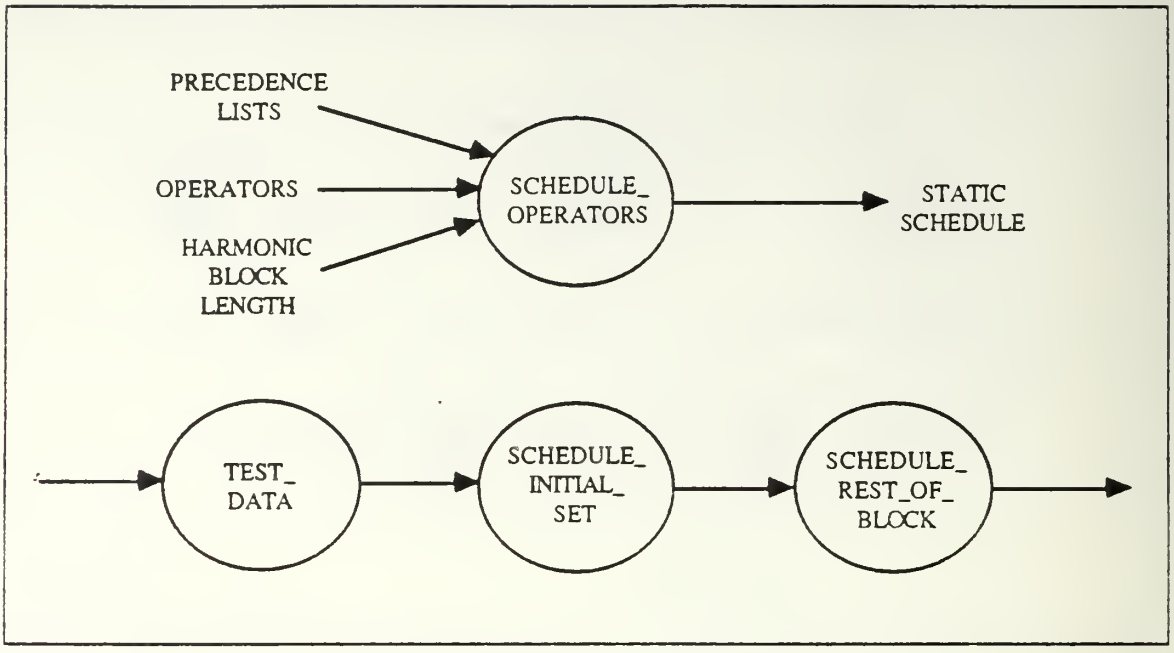


Figure 34. 2nd Level DFD "Schedule_Operators"

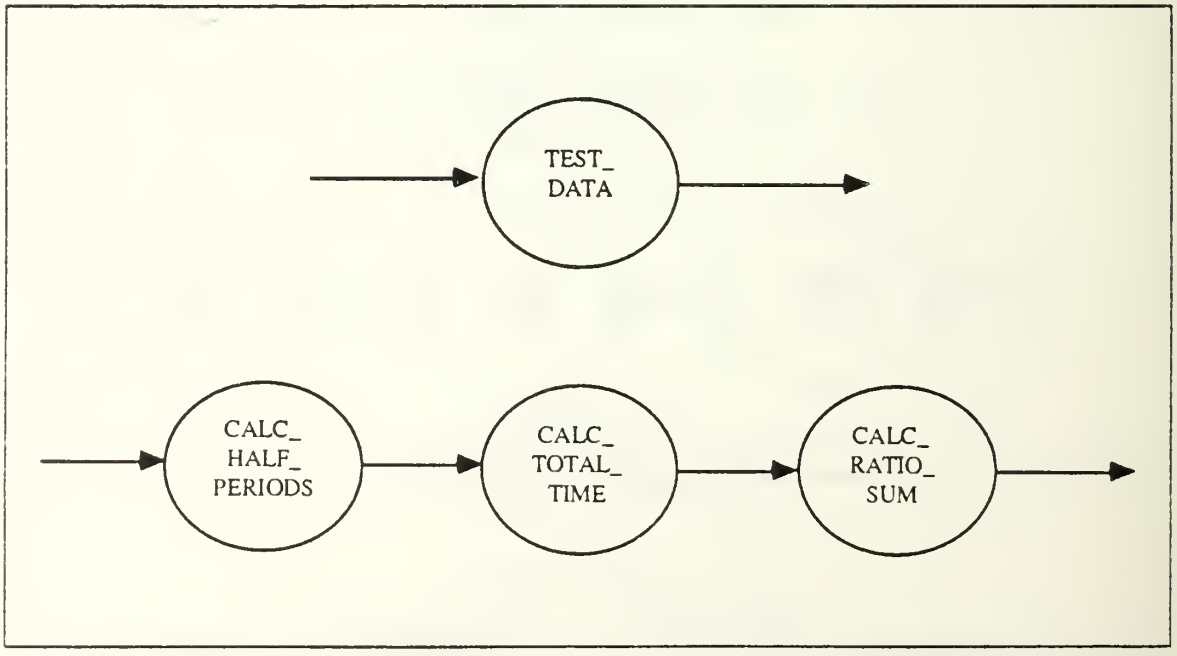


Figure 35. 3rd Level DFD "Test_Data"

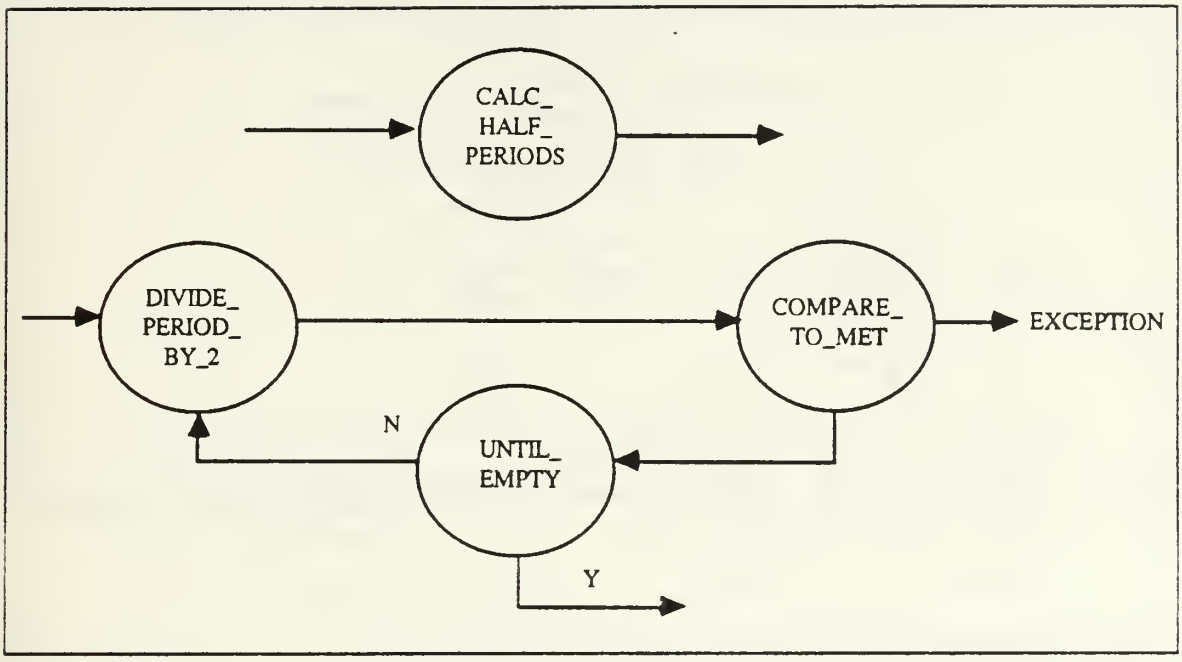


Figure 36. 4th Level DFD "Calc_Half_Periods"

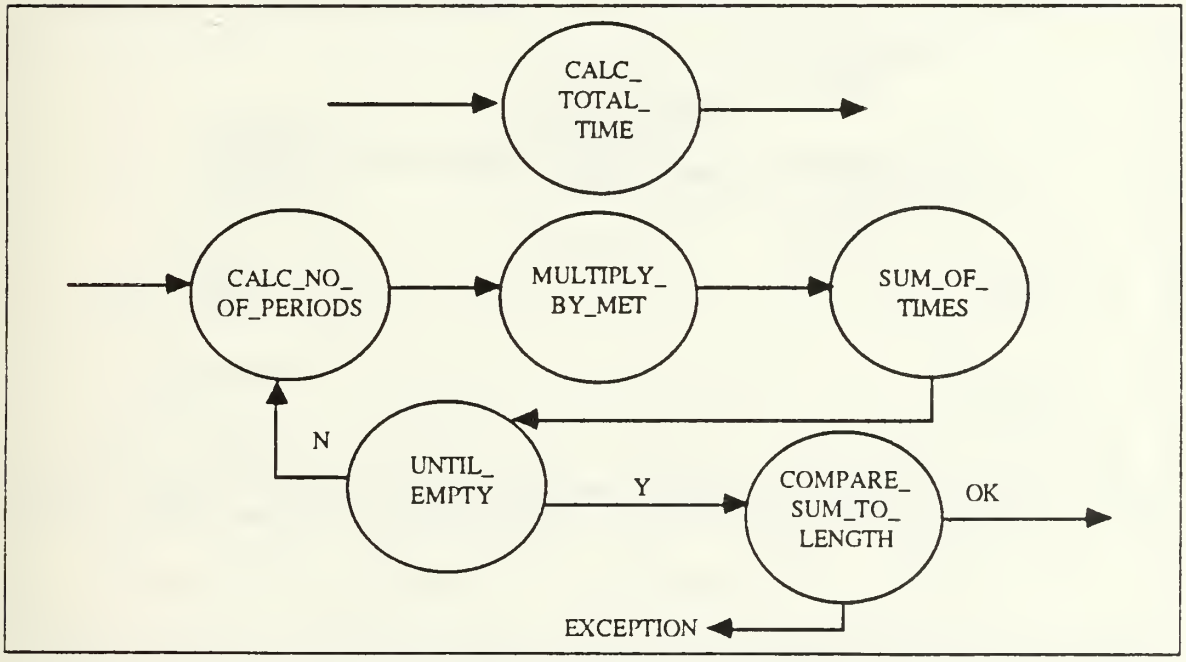


Figure 37. 4th Level DFD "Calc_Total_Time"

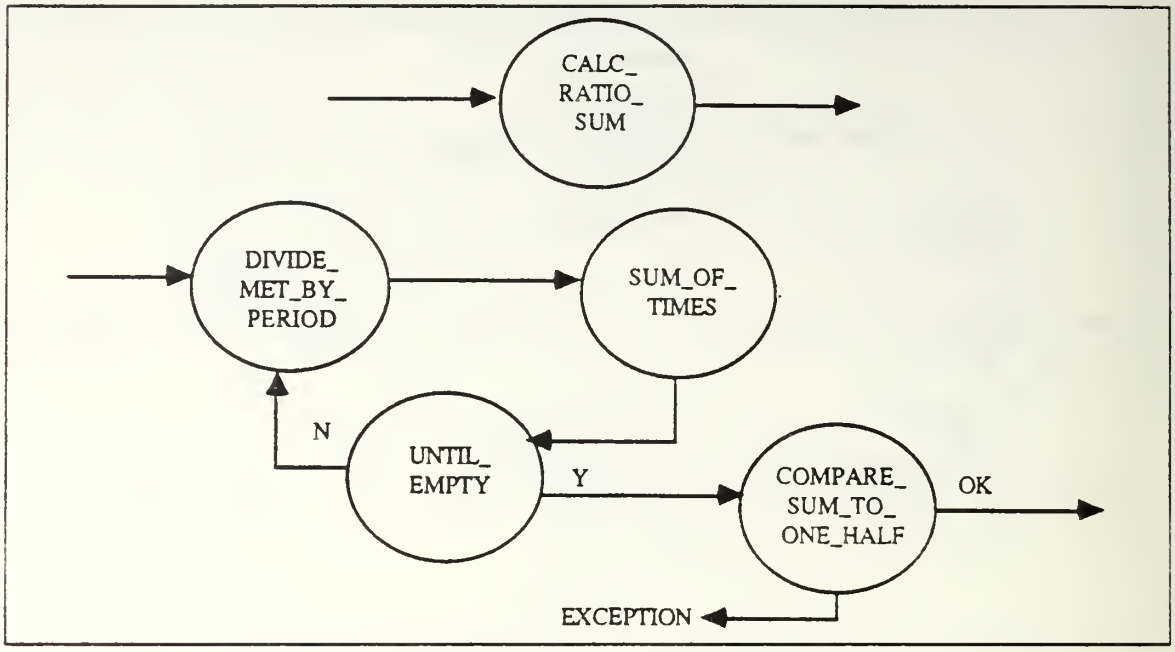


Figure 38. 4th Level DFD "Calc_Ratio_Sum"

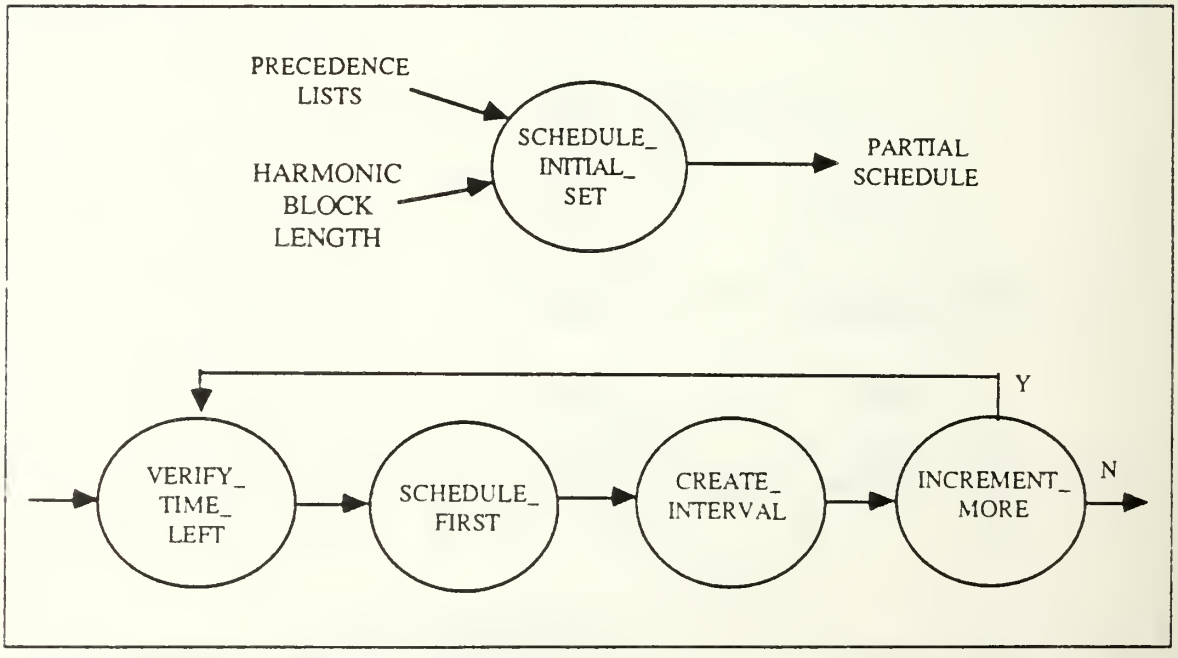


Figure 39. 3rd Level DFD "Schedule_Initial_Set"

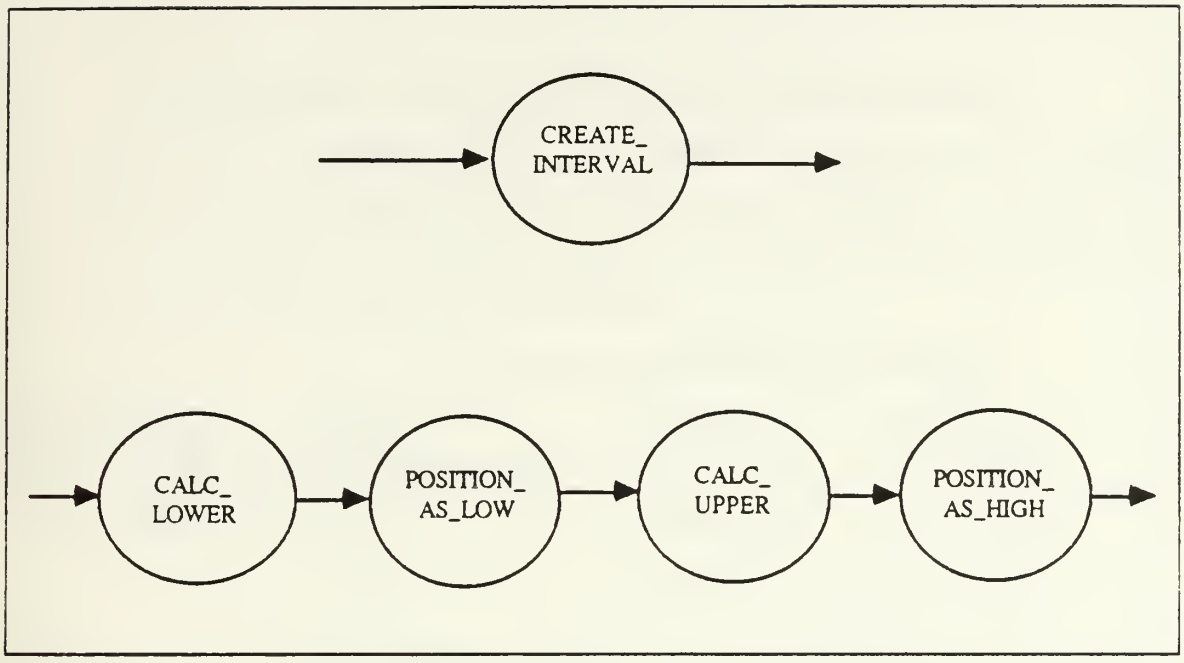


Figure 40. 4th Level DFD "Create_Interval"

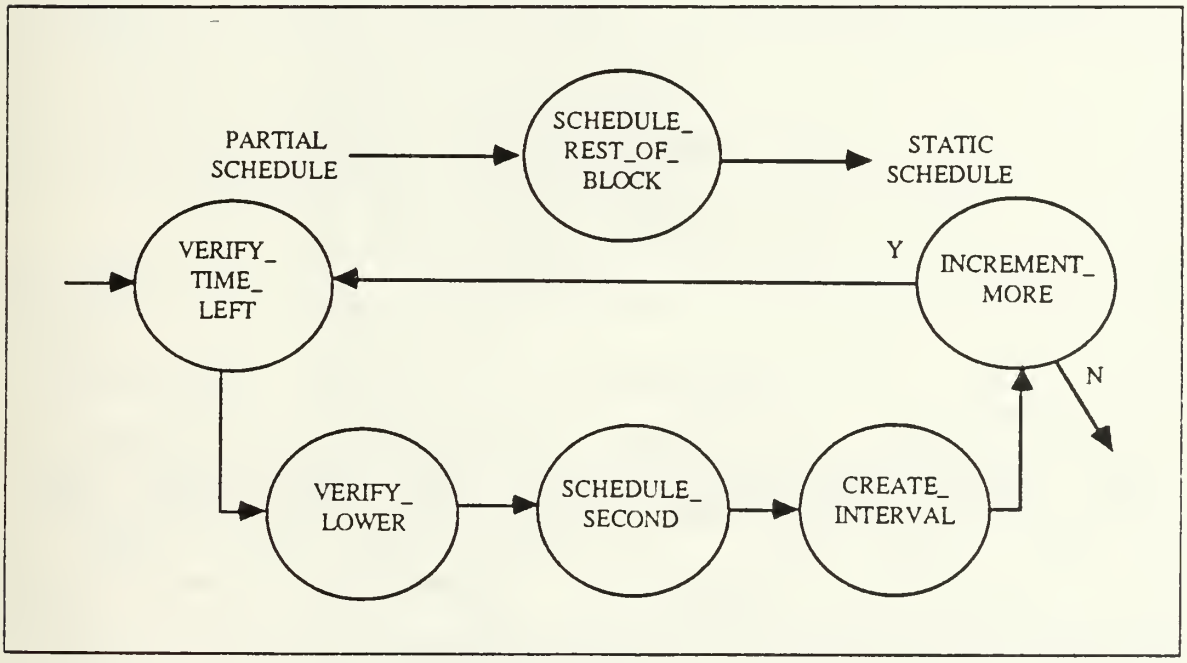


Figure 41. 3rd Level DFD "Schedule_Rest_of_Block"

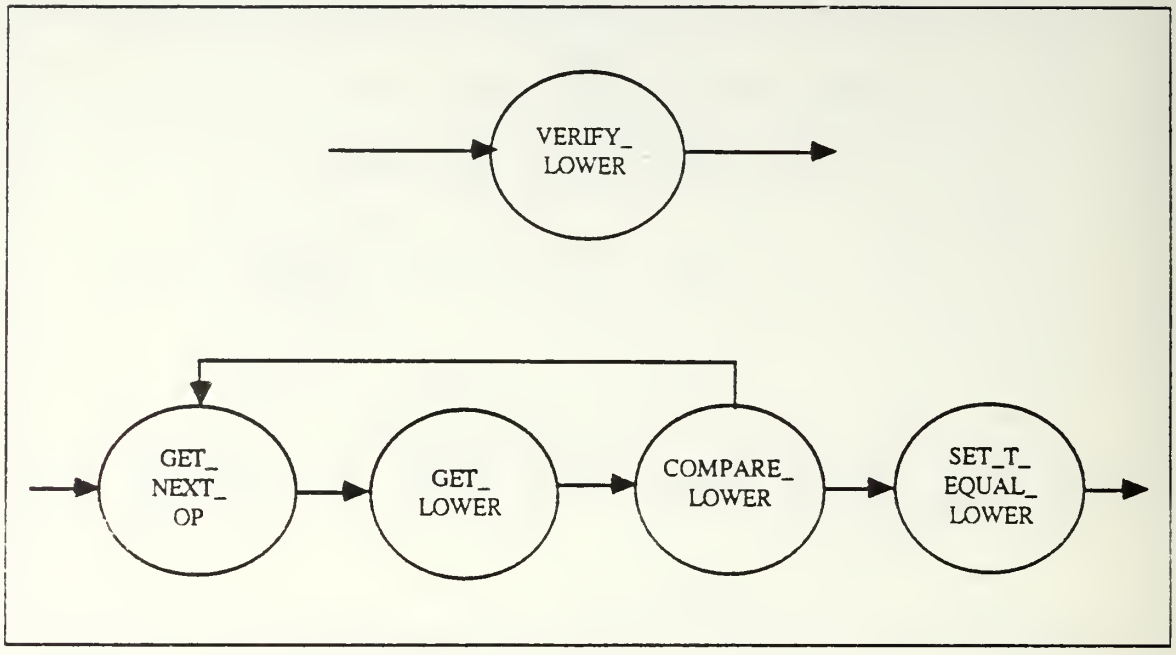


Figure 42. 4th Level DFD "Verify_Lower"

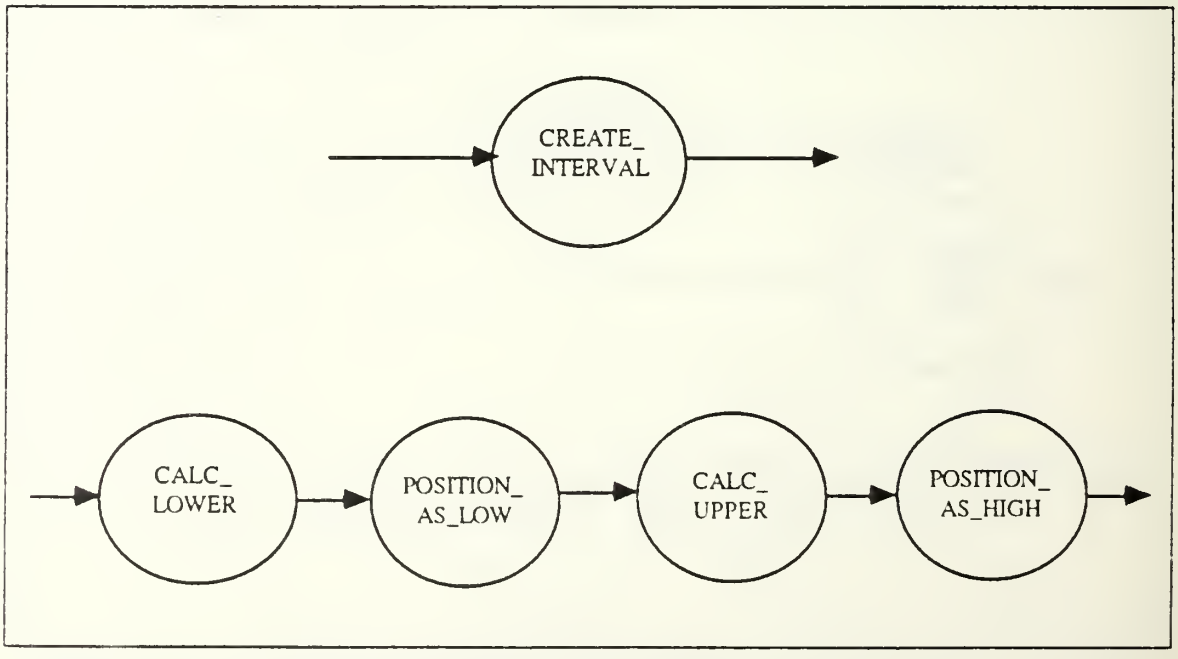


Figure 43. 4th Level DFD "Create_Interval"

APPENDIX D. AG PROCESSOR SOURCE PROGRAM

The following AG Processor source code represents an adaptation of the Kodiyak AG translator generator specifically designed for the Static Scheduler. Its primary objective is identification and retrieval of only the critical operators and their timing constraints from the PSDL prototype source program.

```
%define Digit     :[0-9]
%define Int       :{Digit}+
%define Letter    :[a-zA-Z_]
%define Alpha     :({Letter}|{Digit})
%define Blank     :[ \t\n]
%define Char      :[^{}]

                  :{Blank}+

TYPE              :type|TYPE
OPERATOR          :operator|OPERATOR
SPECIFICATION     :specification|SPECIFICATION
END               :end|END
GENERIC           :generic|GENERIC
INPUT             :input|INPUT
OUTPUT            :output|OUTPUT
STATES            :states|STATES
INITIALLY         :initially|INITIALLY
EXCEPTIONS        :exceptions|EXCEPTIONS
MAX_EXEC_TIME     :maximum[ ]execution[ ]time
                  |MAXIMUM[ ]EXECUTION[ ]TIME
MAX_RESP_TIME     :maximum[ ]response[ ]time
                  |MAXIMUM[ ]RESPONSE[ ]TIME
MIN_CALL_PERIOD   :minimum[ ]calling[ ]period
                  |MINIMUM[ ]CALLING[ ]PERIOD
BY                :by[ ]requirements|BY[ ]REQUIREMENTS
KEYWORDS          :keywords|KEYWORDS
DESCRIPTION       :description|DESCRIPTION
AXIOMS            :axioms|AXIOMS
TEXT              :{Char}*
IMPLEMENTATION    :implementation|IMPLEMENTATION
GRAPH             :graph|GRAPH
DATA_STREAM       :data[ ]stream|DATA[ ]STREAM
TIMER             :timer|TIMER
CONTROL           :control[ ]constraints|CONTROL[ ]CONSTRAINTS
TRIGGERED         :triggered|TRIGGERED
IF                :if|IF
PERIOD            :period|PERIOD
FINISH            :finish[ ]within|FINISH[ ]WITHIN
OUTPUT            :output|OUTPUT
EXCEPTION         :exception|EXCEPTION
RESET             :reset|RESET
START             :start|START
```

```
STOP                : stop|STOP
ALL                 : by[ ]all|BY[ ]ALL
SOME                : by[ ]some|BY[ ]SOME
MICROSEC            : microsec|MICROSEC
MS                  : ms|MS
SEC                 : sec|SEC
MIN                 : min|MIN
HOURS               : hours|HOURS
ADA                 : ada|Ada|ADA
ARROW               : "->"

TRUE                : true|TRUE
FALSE               : false|FALSE

AND                 : "&"|"and"|"AND"
OR                  : "|"|"or"|"OR"
NOT                 : "~"|"not"|"NOT"

ID                  : {Letter}{Alpha}*
INTEGER_LITERAL     : {Int}
REAL_LITERAL        : {Int}"."{Int}

! operator precedences

%left               '*';
%left               OR;
%left               AND;
%left               NOT;
%%

! attribute declarations for nonterminal symbols

start { trn: string; };
psdl { trn: string; };
component { trn: string; };
data_type { trn: string; };
operator { trn: string; };
type_spec { trn: string; };
optional_type_decl { trn: string; };
op_list { trn: string; };
operator_spec { trn: string; };
interface { trn: string; };
attribute { trn: string; };
attribute_list { trn: string; };
generic_param { trn: string; };
input { trn: string; };
output { trn: string; };
states { trn: string; };
exceptions { trn: string; };
id_list { trn: string; };
max_exec_time { trn: string; };
max_resp_time { trn: string; };
min_period { trn: string; };
time { value: int;
       trn: string; };
unit { trn: string; };
```

```
reqmts_trace { trn: string; };
functionality { trn: string; };
keywords { trn: string; };
informal_desc { trn: string; };
formal_desc { trn: string; };
type_impl { trn: string; };
op_impl_list { trn: string; };
operator_impl { trn: string; };
psdl_impl { trn: string; };
data_flow_diagram { trn: string; };
links { trn: string; };
streams { trn: string; };
type_decl { trn: string; };
type_name { trn: string; };
timers { trn: string; };
control_constraints { trn: string; };
constraints { trn: string; };
constraint { trn: string; };
trigger { trn: string; };
cond { trn: string; };
period { trn: string; };
finish { trn: string; };
outputs { trn: string; };
exception_ops { trn: string; };
timer_ops { trn: string; };
timer_op { trn: string; };
opt_trigger_constraint { trn: string; };
predicate { trn: string; };
expression { trn: string; };
expression_list { trn: string; };
op_id { trn: string; };
opt_reqmts_trace { trn: string; };
opt_unit { value: int;
           trn: string; };
opt_keywords { trn: string; };
opt_formal_desc { trn: string; };
opt_informal_desc { trn: string; };
opt_streams { trn: string; };
opt_timers { trn: string; };
opt_control_constraints { trn: string; };
opt_time { trn: string; };
opt_trigger { trn: string; };
opt_cond { trn: string; };
opt_period { trn: string; };
opt_finish { trn: string; };

!attrbute declarations for terminal symbols

ID { %text: string; };
INTEGER_LITERAL { %text: string; };
REAL_LITERAL { %text: string; };

%%
!psdl grammar

start
```

```
        : psdl
          { %output(psdl.trn); }
        ;

psdl
        : psdl component
          { psdl[1].trn = [ psdl[2].trn,component.trn ]; }
        |
          { psdl[1].trn = ""; }
        ;

component

        : data_type
          { component.trn = ""; }
        | operator
          { component.trn = operator.trn; }
        ;

data_type
        : TYPE ID type_spec type_impl
          { data_type.trn = ""; }
        ;

operator
        : OPERATOR ID operator_spec operator_impl
          { operator.trn = ["ID ",ID.%text,operator_spec.trn,
                            operator_impl.trn]; }
        ;

type_spec
        : SPECIFICATION optional_type_decl op_list functionality END
          { type_spec.trn = [ optional_type_decl.trn,op_list.trn,
                              functionality.trn ]; }
        ;

optional_type_decl
        : type_decl
          { optional_type_decl.trn = type_decl.trn; }
        |
          { optional_type_decl.trn = ""; }
        ;

op_list
        : op_list OPERATOR ID operator_spec
          { op_list[1].trn = ""; }
        |
          { op_list[1].trn = ""; }
        ;

operator_spec
        : SPECIFICATION interface functionality END
          { operator_spec.trn = [ "SPEC ",interface.trn,"END " ]; }
        ;
```

```
interface
        : attribute_list
          { interface[1].trn = attribute_list.trn; }
        ;

attribute_list
        : attribute_list attribute opt_reqmts_trace
          { attribute_list[1].trn = [attribute_list[2].trn,
                                     attribute.trn,opt_reqmts_trace.trn]; }
        |
          { attribute_list.trn = ""; }
        ;

opt_reqmts_trace
        : reqmts_trace
          { opt_reqmts_trace.trn = reqmts_trace.trn; }
        |
          { opt_reqmts_trace.trn = ""; }
        ;

attribute
        : generic_param
          { attribute.trn = ""; }
        | input
          { attribute.trn = ""; }
        | output
          { attribute.trn = ""; }
        | states
          { attribute.trn = states.trn; }
        | exceptions
          { attribute.trn = ""; }
        | max_exec_time
          { attribute.trn = max_exec_time.trn; }
        | max_resp_time
          { attribute.trn = max_resp_time.trn; }
        | min_period
          { attribute.trn = min_period.trn; }
        ;

generic_param
        : GENERIC type_decl
          { generic_param.trn = type_decl.trn; }
        ;

input
        : INPUT type_decl
          { input.trn = type_decl.trn; }
        ;

output
        : OUTPUT type_decl
          { output.trn = type_decl.trn; }
        ;

states
        : STATES type_decl INITIALLY expression_list
```

```
          { states.trn = [ "STATE ",type_decl.trn ]; }
        ;

exceptions
        : EXCEPTIONS id_list
          { exceptions.trn = [ "JUNK ",id_list.trn ]; }
        ;

id_list
        : id_list ',' ID
          { id_list[1].trn = ""; }
        | ID
          { id_list[1].trn = ID.%text; }
        ;

max_exec_time
        : MAX_EXEC_TIME time
          { max_exec_time.trn = ["MET ",i2s(time.trn)]; }
        ;

max_resp_time
        : MAX_RESP_TIME time
          { max_resp_time.trn = ["MRT ",i2s(time.trn)]; }
        ;

min_period
        : MIN_CALL_PERIOD time
          { min_period.trn = ["MCP ",i2s(time.trn)]; }
        ;

time
        : INTEGER_LITERAL opt_unit
          { time.trn = s2i(INTEGER_LITERAL.%text) * opt_unit.value;
                       time.trn = i2s (time.value); }
        ;

opt_unit
        : unit
          { opt_unit.trn = unit.trn; }
        |
          { opt_unit.trn = ""; }
        ;

unit
        : MICROSEC
          { unit.trn = "1"; }
        | MS
          { unit.trn = "1000"; }
        | SEC
          { unit.trn = "1000000"; }
        | MIN
          { unit.trn = "60000000"; }
        | HOURS
          { unit.trn = "3600000000"; }
        ;
```

```
reqmts_trace
        : BY id_list
          { reqmts_trace.trn = [ "JUNK ",id_list.trn ]; }
        ;

functionality
        : opt_keywords opt_informal_desc opt_formal_desc
          { functionality.trn = [ opt_keywords.trn,opt_informal_desc.trn,
                                  opt_formal_desc.trn ]; }
        ;

opt_keywords
        : keywords
          { opt_keywords.trn = keywords.trn; }
        |
          { opt_keywords.trn = ""; }
        ;

opt_informal_desc
        : informal_desc
          { opt_informal_desc.trn = informal_desc.trn; }
        |
          { opt_informal_desc.trn = ""; }
        ;

opt_formal_desc
        : formal_desc
          { opt_formal_desc.trn = formal_desc.trn; }
        |
          { opt_formal_desc.trn = ""; }
        ;

keywords
        : KEYWORDS id_list
          { keywords.trn = [ "JUNK ",id_list.trn ]; }
        ;

informal_desc
        : DESCRIPTION '{' TEXT '}'
          { informal_desc.trn = ""; }
        ;

formal_desc
        : AXIOMS '{' TEXT '}'
          { formal_desc.trn = ""; }
        ;

type_impl
        : IMPLEMENTATION ADA ID
          { type_impl.trn = ""; }
        | IMPLEMENTATION type_name op_impl_list END
          { type_impl.trn = [type_name.trn,op_impl_list.trn];}
        ;

op_impl_list
        : op_impl_list OPERATOR ID operator_impl
```

```
            { op_impl_list[1].trn =
                              [ "JUNK ",operator_impl.trn [; }
          |
            { op_impl_list[1].trn = ""; }
          ;

operator_impl
          : IMPLEMENTATION ADA ID
            { operator_impl.trn = "ADA "; }
          | IMPLEMENTATION psdl_impl
            { operator_impl.trn = ["IMPL ",psdl_impl.trn]; }
          ;

psdl_impl
          : data_flow_diagram opt_streams opt_timers opt_control_constraints
            opt_informal_desc END
            { psdl_impl.trn = [data_flow_diagram.trn,opt_streams.trn,
                               opt_timers.trn,opt_informal_desc.trn,
                               opt_control_constraints.trn,"END "]; }
          |
            { psdl_impl.trn = ""; }
          ;

data_flow_diagram
          : GRAPH links
            { data_flow_diagram.trn = links.trn; }
          ;

links
          : links ID '.' op_id ARROW ID
            { links[1].trn = ["LINK ",ID.%text," . ", op_id.trn,
                              "ARROW ",ID.%text]; }
          |
            { links[1].trn = ""; }
          ;

op_id
          : ID ':' opt_time
            { op_id.trn = [ID.%text," : ",opt_time.trn]; }
          | ID
            { op_id.trn = ID.%text; }
          ;

opt_time
          : time
            { opt_time.trn = time.trn; }
          |
            { opt_time.trn = ""; }
          ;

opt_streams
          : streams
            { opt_streams.trn = streams.trn; }
          |
            { opt_streams.trn = ""; }
          ;
```

```
opt_timers
        : timers
          { opt_timers.trn = timers.trn; }
        |
          { opt_timers.trn = ""; }
        ;

opt_control_constraints
        : control_constraints
          { opt_control_constraints.trn = control_constraints.trn; }
        |
          { opt_control_constraints.trn = ""; }
        ;

streams
        : DATA_STREAM type_decl
          { streams.trn = type_decl.trn; }
        ;

type_decl
        : type_decl ',' id_list ':' type_name
          { type_decl[1].trn = [ "JUNK ",id_list.trn ]; }
        | id_list ':' type_name
          { type_decl[1].trn = [ "JUNK ",id_list.trn ]; }
        ;

type_name
        : ID
          { type_name.trn = ""; }
        | ID '[' type_decl ']'
          { type_name.trn = ""; }
        ;

timers
        : TIMER id_list
          { timers.trn = [ "JUNK ",id_list.trn ]; }
        ;

control_constraints
        : CONTROL constraints
          { control_constraints.trn = constraints.trn; }
        ;

constraints
        : constraints constraint
          { constraints[1].trn = [ constraints[2].trn,
                                   constraint.trn ]; }
        |
          { constraints.trn = ""; }
        ;

constraint
        : OPERATOR ID opt_trigger_constraint opt_period
          opt_finish outputs exception_ops timer_ops
          { constraint[1].trn = ["ID ",ID.%text,opt_trigger_constraint.trn,
```

```
                                        opt_period.trn,opt_finish.trn,outputs.trn,
                                        exception_ops.trn,timer_ops.trn ]; }
        ;

opt_trigger
        : trigger
          { opt_trigger.trn = trigger.trn; }
        |
          { opt_trigger.trn = ""; }
        ;

opt_period
        : period
          { opt_period.trn = period.trn; }
        |
          { opt_period.trn = ""; }
        ;

opt_finish
        : finish
          { opt_finish.trn = finish.trn; }
        |
          { opt_finish.trn = ""; }
        ;

opt_cond
        : cond
          { opt_cond.trn = cond.trn; }
        |
          { opt_cond.trn = ""; }
        ;

opt_trigger_constraint
        : TRIGGERED opt_trigger opt_cond opt_reqmts_trace
          { opt_trigger_constraint.trn = [ opt_trigger.trn,opt_cond.trn,
                                           opt_reqmts_trace.trn ]; }
        |
          { opt_trigger_constraint.trn = ""; }
        ;

cond
        : IF predicate
         { cond.trn = predicate.trn; }
        ;

period
        :PERIOD time opt_reqmts_trace
         { period.trn = ["PERIOD ",time.trn,opt_reqmts_trace.trn]; }
        ;

finish
        : FINISH time opt_reqmts_trace
         { finish.trn = ["FINISH ",time.trn,opt_reqmts_trace.trn]; }
        ;

outputs
```

```
        : outputs OUTPUT id_list cond opt_reqmts_trace
          { outputs[1].trn = [ "JUNK ",id_list.trn,cond.trn,
                               opt_reqmts_trace.trn ]; }
        |
          { outputs[1].trn = ""; }
        ;

exception_ops
        : exception_ops EXCEPTION ID opt_cond opt_reqmts_trace
          { exception_ops[1].trn = [ opt_cond.trn,
                                     opt_reqmts_trace.trn ]; }
        |
          { exception_ops[1].trn = ""; }
        ;

timer_ops
        : timer_ops timer_op ID opt_cond opt_reqmts_trace
          { timer_ops[1].trn = [ timer_op.trn,opt_cond.trn,
                                 opt_reqmts_trace.trn ]; }
        |
          { timer_ops[1].trn = ""; }
        ;

timer_op
        : RESET
          { timer_op.trn = ""; }
        | START
          { timer_op.trn = ""; }
        | STOP
          { timer_op.trn = ""; }
        ;

trigger
        : ALL id_list
          { trigger.trn = [ "JUNK ",id_list.trn ]; }
        | SOME id_list
          { trigger.trn = [ "JUNK ",id_list.trn ]; }
        ;

predicate
        : NOT predicate                 %prec NOT
          { predicate[1].trn = ""; }
        | predicate AND predicate       %prec AND
          { predicate[1].trn = ""; }
        | predicate OR predicate        %prec OR
          { predicate[1].trn = ""; }
        | expression
          { predicate[1].trn = expression.trn; }
        | ID ':' id_list
          { predicate[1].trn = id_list.trn; }
        ;

expression
        : INTEGER_LITERAL
          { expression.trn = ""; }
        | TRUE
```

```
          { expression.trn = ""; }
        | FALSE
          { expression.trn = ""; }
        | ID
          { expression.trn = ""; }
        | type_name '.' ID '(' expression_list ')'
          { expression.trn = ""; }
        ;

expression_list
        : expression_list ',' expression
          { expression_list[1].trn = ""; }
        | expression
          { expression_list[1].trn = ""; }
        ;
```

APPENDIX E. STATIC SCHEDULER PSEUDOCODE LISTING

The following Static Scheduler pseudocode represents a guideline for implementing an executable Ada® program. Precise naming conventions for program units, file and data types, and Ada® EXCEPTIONS specifically relate to the Static Scheduler. Chapters III and IV of this thesis outline the underlying assumptions and concerns that were considered during development of this pseudocode.

```
package FILES is

  type DATA_STREAM is new STRING ( 1..40 );
  type OPERATOR_ID is new STRING ( 1..40 );
  type VALUE is new STRING ( 0..10 );
  type MET is VALUE;
  type MRT is VALUE;
  type MCP is VALUE;
  type PERIOD is VALUE;
  type WITHIN is VALUE;
  type STARTS is VALUE;
  type STOPS is VALUE;
  type LOWERS is VALUE;
  type UPPERS is VALUE;

  type LINKS_is
    record
      THE_DATA_STREAM  :  DATA_STREAM;
      THE_FIRST_OP_ID  :  OPERATOR_ID;
      THE_LINK_MET     :  MET  := 0;
      THE_SECOND_OP_ID :  OPERATOR_ID;
    end record;

  type OPERATORS is
    record
      THE_OPERATOR_ID  : OPERATOR_ID;
      THE_MET          : MET  := 0;
      THE_MRT          : MRT  := 0;
      THE_MCP          : MCP  := 0;
      THE_PERIOD       : PERIOD  := 0;
      THE_WITHIN       : WITHIN  := 0;
    end record;

  type PRECEDENCE_LIST is
    record
      THE_LEFT_OP_ID   :  OPERATOR_ID;
      THE_RIGHT_OP_ID  :  OPERATOR_ID;
    end record;

  type SCHEDULE_INPUTS is
    record
```

```
      OPERATOR          : OPERATOR_ID;
      THE_START         : STARTS  := 0;
      THE_STOP          : STOPS   := 0;
      THE_LOWER         : LOWERS  := 0;
      THE_UPPER         : UPPERS  := 0;
    end record;
end FILES;



with TEXT_IO;
package PSDL_READER is

  procedure INVOKE_AG_PROCESSOR ( PSDL_PROG : TEXT_IO.IN_FILE;
                                  AG_OUTFILE : TEXT_IO.OUT_FILE );
  procedure READ_THE_FILE ( AG_OUTFILE : TEXT_IO.OUT_FILE );

end PSDL_READER;



with TEXT_IO;
with FILES;
package FILE_PROCESSOR is

  procedure SEPARATE_DATA (AG_OUTFILE : TEXT_IO.IN_FILE;
                           LINKS      : TEXT_IO.OUT_FILE;
                           OPERATORS  : TEXT_IO.OUT_FILE;
                           NON_CRITS  : TEXT_IO.OUT_FILE );
  procedure VALIDATE_DATA ( OPERATORS : TEXT_IO.IN_FILE );

  MET_EQUALS_PERIOD    : exception;
  MET_NOT_LESS_THAN_MRT : exception;

end FILE_PROCESSOR;



with FILES;
with TEXT_IO;
package TOPOLOGICAL_SORTER is

  procedure CREATE_LISTS ( LINKS           : TEXT_IO.IN_FILE;
                           PRECEDENCE_LIST : TEXT_IO.OUT_FILE );
  procedure SORT_REMAINING_OPERATORS ( PRECEDENCE_LIST :
                                       TEXT_IO.OUT_FILE;
                                     LINKS : TEXT_IO.IN_FILE );

  NO_INITIAL_LINK_OP : exception;
  NO_MATCHES_FOUND   : exception;

end TOPOLOGICAL_SORTER;
```

```
with FILES;
package HARMONIC_BLOCK_BUILDER is

   procedure CALC_PERIODIC_EQUIVALENTS ( OPERATORS : TEXT_IO.IN_FILE );
   procedure SORT_BY_PERIOD ( THE_PERIOD : in out OPERATORS );
   procedure FIND_BASE_BLOCK ( THE_PERIOD : in out OPERATORS;
                               BASE_BLOCK : out INTEGER );
   procedure FIND_BLOCK_LENGTH ( BASE_BLOCK : in out INTEGER;
                                 HARMONIC_BLOCK_LENGTH : out INTEGER );

   CONSTRAINTS_INVALID : exception;
   NO_BASE_BLOCK       : exception;

end HARMONIC_BLOCK_BUILDER;



with FILES;
package OPERATOR_SCHEDULER is

   procedure TEST_DATA
   procedure SCHEDULE_INITIAL_SET ( PRECEDENCE_LIST : TEXT_IO.IN_FILE;
                                    SCHEDULE_INPUTS : TEXT_IO.OUT_FILE;
                                    CONTINUE        : in out BOOLEAN;
                                    THE_MET         : in out OPERATORS );
   procedure SCHEDULE_REST_OF_BLOCK ( SCHEDULE_INPUTS : TEXT_IO.OUT_FILE;
                                      THE_MET         : in out OPERATORS );
   procedure CREATE_STATIC_SCHEDULE (SCHEDULE_INPUTS : TEXT_IO.IN_FILE;
                                     STATIC_SCHEDULE : TEXT_IO.OUT_FILE);


   FAIL_HALF_PERIOD  :  exception;
   BAD_TOTAL_TIME    :  exception;
   RATIO_TOO_BIG     :  exception;
   OVER_TIME         :  exception;
   INVALID_SCHEDULE  :  exception;
   SCHEDULE_ERROR    :  exception;

end OPERATOR_SCHEDULER;



with TEXT_IO;
with FILES;
with PSDL_READER;
with FILE_PROCESSOR;
with TOPOLOGICAL_SORTER;
with HARMONIC_BLOCK_BUILDER;
with OPERATOR_SCHEDULER;

procedure STATIC_SCHEDULER is

   task INITIATE_STATIC_SCHEDULER;

begin
```

```
task body INITIATE_STATIC_SCHEDULER is
begin
  accept STATIC;
end INITIATE_STATIC_SCHEDULER;

PSDL_READER.INVOKE_AG_PROCESSOR;
PSDL_READER.READ_THE_FILE;
FILE_PROCESSOR.SEPARATE_DATA;
FILE_PROCESSOR.VALIDATE_DATA;
TOPOLOGICAL_SORTER.CREATE_LISTS;
TOPOLOGICAL_SORTER.SORT_REMAINING_OPERATORS;
while not END_OF_FILE loop
  HARMONIC_BLOCK_BUILDER.CALC_PERIODIC_EQUIVALENTS;
end loop;
HARMONIC_BLOCK_BUILDER.SORT_BY_PERIOD;
HARMONIC_BLOCK_BUILDER.FIND_BASE_BLOCKS;
HARMONIC_BLOCK_BUILDER.FIND_BLOCK_LENGTH;
OPERATOR_SCHEDULER.TEST_DATA;
OPERATOR_SCHEDULER.SCHEDULE_INITIAL_SET;
OPERATOR_SCHEDULER.SCHEDULE_REST_OF_BLOCK;
OPERATOR_SCHEDULER.CREATE_STATIC_SCHEDULE;

exception
  when FILE_PROCESSOR.MET_NOT_LESS_THAN_MRT =>
    PUT_LINE ( "Cannot allow MET larger than or equal to MRT." );
exception
  when FILE_PROCESSOR.MET_EQUALS_PERIOD =>
    PUT_LINE ( "Cannot allow MET equal to period." );
exception
  when TOPOLOGICAL_SORTER.NO_INITIAL_LINK_OP =>
    PUT_LINE ( "Cannot locate the initial link statement." );
exception
  when TOPOLOGICAL_SORTER.NO_MATCHES_FOUND =>
    PUT_LINE ( "Cannot locate any link statements after the first." );
exception
  when HARMONIC_BLOCK_BUILDER.CONSTRAINTS_INVALID =>
    PUT_LINE ( "Cannot build schedule with given constraints." );
exception
  when HARMONIC_BLOCK_BUILDER.NO_BASE_BLOCK =>
    PUT_LINE ( "Cannot schedule incompatible operators." );
exception
  when OPERATOR_SCHEDULER.FAIL_HALF_PERIOD =>
    PUT_LINE ( "Cannot guarantee schedule will be feasible." );
exception
  when OPERATOR_SCHEDULER.BAD_TOTAL_TIME =>
    PUT_LINE ( "Cannot guarantee schedule will be feasible." );
exception
  when OPERATOR_SCHEDULER.RATIO_TOO_BIG =>
    PUT_LINE ( "Cannot guarantee schedule will be feasible." );
exception
  when OPERATOR_SCHEDULER.OVER_TIME =>
    PUT_LINE ( "Cannot create schedule with parameters given." );
exception
  when OPERATOR_SCHEDULER.INVALID_SCHEDULE =>
    PUT_LINE ( "Cannot create schedule with parameters given." );
exception
```

```
    when OPERATOR_SCHEDULER.SCHEDULE_ERROR =>
      PUT_LINE ( "Cannot create schedule with parameters given." );

end STATIC_SCHEDULER;



with TEXT_IO;
package body PSDL_READER is

  A_WORD : STRING;

  procedure INVOKE_AG_PROCESSOR ( PSDL_PROG : TEXT_IO.IN_FILE;
                                  AG_OUTFILE : TEXT_IO.OUT_FILE );
  begin
  --  Run the compiled AG program
  --  using the PSDL_PROG as its input
  --  to create the AG_OUTFILE as its output

  end INVOKE_AG_PROCESSOR;



  procedure READ_THE_FILE ( AG_OUTFILE : TEXT_IO.IN_FILE;

  TRASH_FILE : TEXT_IO.OUT_FILE;

  begin
    OPEN ( AG_OUTFILE, TEXT_IO.IN_FILE );
    CREATE ( TRASH_FILE, TEXT_IO.OUT_FILE );
    while not TEXT_IO.END_OF_FILE loop
      GET ( A_WORD[i] : out AG_OUTFILE );
      if A_WORD[i] = "JUNK " then
        begin
          PUT ( A_WORD[i]   : in TRASH_FILE );
          PUT ( A_WORD[i+1] : in TRASH_FILE );
        end;
      end if;
      A_WORD[i] := A_WORD[i+1];
    end loop;

  end READ_THE_FILE;

end PSDL_READER;



with TEXT_IO;
package body FILE_PROCESSOR is

  procedure SEPARATE_DATA ( AG_OUTFILE : TEXT_IO.IN_FILE;
                            LINKS      : TEXT_IO.OUT_FILE;
                            OPERATORS  : TEXT_IO.OUT_FILE;
                            NON_CRITS  : TEXT_IO.OUT_FILE ) is
  NEW_WORD  : STRING;
  THE_STATE : STRING;
```

```
begin
  OPEN (AG_OUTFILE, TEXT_IO.IN_FILE );
  CREATE ( LINKS, TEXT_IO.OUT_FILE );
  CREATE ( OPERATORS, TEXT_IO.OUT_FILE ):

  while not TEXT_IO.END_OF_FILE loop
    GET ( NEW_WORD[i] : out AG_OUTFILE );
    if NEW_WORD[i] = "ID " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_OPERATOR_ID );
    elsif NEW_WORD[i] = "STATE " then
      THE_STATE := NEW_WORD[i+1]
    elsif NEW_WORD[i] = "MET " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_MET );
    elsif NEW_WORD[i] = "MRT " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_MRT );
    elsif NEW_WORD[i] = "MCP " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_MCP );
    elsif NEW_WORD[i] = "PERIOD " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_PERIOD );
    elsif NEW_WORD[i] = "WITHIN " then
      PUT ( NEW_WORD[i+1] : in OPERATORS.THE_WITHIN );
    elsif NEW_WORD[i] = "LINK " then
      begin
        if NEW_WORD[i+3] = THE_STATE and
           NEW_WORD[i+7] = THE_STATE then

          begin
            -- Discard this link statement since
            -- it represents a state machine.
            -- Increment NEW_WORD.
          end;
        else
          PUT ( NEW_WORD[i+1] : in LINKS.THE_DATA_STREAM );
          PUT ( NEW_WORD[i+3] : in LINKS.THE_FIRST_OP_ID );
          if NEW_WORD[i+4] := " : " then
            PUT ( NEW_WORD[i+5] : in LINKS.THE_LINK_MET );
          else
            PUT ( NEW_WORD[i+7] : in LINKS.THE_SECOND_OP_ID );
          end if;
        end if;
      end;
    else NEW_WORD[i] := NEW_WORD[i+2];
    end if;
  end loop;

  CREATE ( NOT_CRITS, TEXT_IO.OUT_FILE );
  while not OPERATORS.END_OF_FILE loop
    if THE_MET[i] = null STRING then
      PUT ( THE_OPERATOR_ID[i] : in NON_CRITS );
    end if;
    THE_MET[i] = THE_MET[i+1];
  end loop;

end SEPARATE_DATA;
```

```
  procedure VALIDATE_DATA ( OPERATORS : TEXT_IO.IN_FILE ) is

  begin
    OPEN ( OPERATORS, TEXT_IO.IN_FILE );
    while not OPERATORS.END_OF_FILE loop
      GET ( THE_MCP[i] : out OPERATORS );
      if THE_MCP[i] not= 0 then
        begin
          GET ( THE_MRT[i] : out OPERATORS );
          if THE_MRT[i] = 0 then
            begin
              GET ( THE_WITHIN[i] : out OPERATORS );
              if THE_WITHIN[i] not= 0 then
                THE_MRT[i] := THE_WITHIN[i];
              end if;
            end;
          else
            GET ( THE_PERIOD[i] : out OPERATORS );
            THE_MRT[i] := THE_PERIOD[i];
          end if;
        end;
      end if;

    -- Two additional "if...then" loops similar to the above would
    -- also appear here within the loop.
    -- (1) Verify that MET not = period,
    --     else raise the exception MET_EQUALS_PERIOD.
    -- (2) Verify that MET < MRT,
    --     else raise the exception MET_NOT_LESS_THAN_MRT.
    end loop;

  end VALIDATE_DATA;

end FILE_PROCESSOR;



with TEXT_IO;
package body TOPOLOGICAL_SORTER is


  procedure CREATE_LISTS ( LINKS : TEXT_IO.IN_FILE;
                           PRECEDENCE_LIST : TEXT_IO.OUT_FILE ) is
  begin
    CREATE ( PRECEDENCE_LIST, TEXT_IO.OUT_FILE );
    OPEN ( LINKS, TEXT_IO.IN_FILE );
    while not LINKS.END_OF_FILE loop
      GET ( THE_FIRST_OP_ID[i] : out LINKS );
      loop THRU_ALL_SECOND_OP_IDS
        GET ( THE_SECOND_OP_ID[j] : out LINKS );
        if THE_FIRST_OP_ID[i] never= THE_SECOND_OP_ID[j] then
          begin
          PUT ( THE_FIRST_OP_ID : in PRECEDENCE_LIST.THE_LEFT_OP_ID );
          PUT ( THE_SECOND_OP_ID : in PRECEDENCE_LIST.THE_RIGHT_OP_ID);
```

```
          end;
        elsif
          THE_FIRST_OP_ID[i] = any of THE_SECOND_OP_ID[j] then
          exit this iteration and try THE_FIRST_OP_ID[i+1];
      end loop;

      exception
        when PRECEDENCE_LIST = null =>
          raise NO_INITIAL_LINK_OP;
          (exit and terminate the Static Scheduler)

  end CREATE_LISTS;

  procedure SORT_REMAINING_OPERATORS ( LINKS : TEXT_IO.IN_FILE;
                                PRECEDENCE_LIST : TEXT_IO.OUT_FILE ) is
  begin
    GET ( THE_RIGHT_OP_ID[i] : out PRECEDENCE_LIST );
    while not PRECEDENCE_LIST.END_OF_FILE loop
      --  Comparisons similar to procedure CREATE_LISTS would
      --  go here but in this case you want to find instances
      --  where THE_RIGHT_OP_ID[i].PRECEDENCE_LIST =
      --        THE_FIRST_OP_ID[i+1].LINKS
      if a match is found then
        THE_LEFT_OP_ID[i+1].PRECEDENCE_LIST :=
          THE_FIRST_OP_ID[i+1].LINKS and
        THE_RIGHT_OP_ID[i+1].PRECEDENCE_LIST :=
        THE_SECOND_OP_ID[i+1].LINKS
      end if;
      THE_RIGHT_OP_ID[i] := THE_RIGHT_OP_ID[i+1];
    end loop;

    exception
      when PRECEDENCE_LIST contains only an initial entry =>
        raise NO_MATCHES_FOUND;
        (exit and terminate the Static Scheduler)

  end SORT_REMAINING_OPERATORS;

end TOPOLOGICAL_SORTER;



package body HARMONIC_BLOCK_BUILDER is


  procedure CALC_PERIODIC_EQUIVALENTS ( OPERATORS : TEXT_IO.IN_FILE ) is

  begin

    procedure LOCATE_SPORADIC_OPERATOR
    begin
      GET ( THE_MCP[i] : out OPERATORS );
      if found
        GET ( THE_PERIOD[i] : out OPERATORS );
        if THE_PERIOD[i] not= 0 then
          begin
```

```
          restart the loop, this is already a periodic operator
          GET ( THE_MCP[i+1] : out OPERATORS );
        end;
      else
        GET ( THE_MRT[i] : out OPERATORS );
        if THE_MRT[i] = 0 then
          begin
            THE_MRT[i] := THE_WITHIN[i] or if no "within" in file
            THE_MRT[i] := THE_MET[i];
          end;
        end if;
      else
        all operators are already periodic
      exit this package
    end if;

  end LOCATE_SPORADIC_OPERATOR;

  procedure VERIFY_1 is
  begin
    GET ( THE_MCP[i] : out OPERATORS );
    GET ( THE_MRT[i] : out OPERATORS );

    exception
      when THE_MCP[i] >= THE_MRT[i] =>
        raise CONSTRAINTS_INVALID;
        (exit and terminate the Static Scheduler)

      --  Two additional checks are structured the same as above
      --  (1) verify that THE_MET[i] < THE_MRT[i]
      --  (2) verify that THE_MET[i] < THE_MCP[i]
      --  both (1) and (2) will raise exception CONSTRAINTS_INVALID

  end VERIFY_1;

  procedure PERIOD_ALGO is

    DIFFERENCE : STRING;
    NEW_PERIOD : STRING;

  begin
    DIFFERENCE := THE_MRT[i] - THE_MET[i];
    if DIFFERENCE < THE_MCP[i] and DIFFERENCE > THE_MET[i] then
      NEW_PERIOD[i] := DIFFERENCE;
    else
      NEW_PERIOD[i] := THE_MCP[i];
    end if;
    PUT ( NEW_PERIOD[i] :  in OPERATORS.THE_PERIOD[i] );

  end PERIOD_ALGO;

end CALC_PERIODIC_EQUIVALENTS;



procedure SORT_BY_PERIOD ( THE_PERIOD :  in out OPERATORS );
```

```
begin
  perform_sort in ascending order based on THE_PERIOD in OPERATORS

end SORT_BY_PERIOD;



procedure FIND_BASE_BLOCKS ( THE_PERIOD : in out OPERATORS;
                             BASE_BLOCK : out INTEGER ) is
  DIVISOR        :  INTEGER;
  REMAINDER      :  INTEGER;
  ORIG_SEQUENCE  :  {};        -- null sequence
  TEMP_SEQUENCE  :  {};        -- null sequence
  BASE_BLOCK     :  {};        -- null sequence

function MOD_DIVIDE ( THE_PERIOD : out ORIG_SEQUENCE ) returns
                                                       REMAINDER;
begin
  loop CREATE_PERIOD_SEQUENCE
    GET ( THE_PERIOD[i] : out OPERATORS );
    PUT ( THE_PERIOD[i] : in ORIG_SEQUENCE );
  end loop CREATE_PERIOD_SEQUENCE;
  loop CREATE_BLOCKS
    GET ( THE_PERIOD[i] : out ORIG_SEQUENCE );
    DIVISOR := THE_PERIOD[i];
    loop INNER

      function MOD_DIVIDE returns REMAINDER is
        REMAINDER := THE_PERIOD[i] / DIVISOR;
        return REMAINDER;
      end MOD_DIVIDE;

      if REMAINDER = 0 then
        PUT ( THE_PERIOD[i] : in BASE_BLOCK );
      else
        PUT ( THE_PERIOD[i] : in TEMP_SEQUENCE );
      end if;
    end loop INNER when the ORIG_SEQUENCE = {};       -- null
    if TEMP_SEQUENCE = {} then return
      -- loop CREATE_PERIOD_SEQUENCE is completed
    elsif
      TEMP_SEQUENCE not= {} and DIVISOR not= 1 then
        begin
          ORIG_SEQUENCE := BASE_BLOCK U TEMP_SEQUENCE;
        end;
    else

      exception
      when TEMP_SEQUENCE not= {} and DIVISOR = 1 =>
        raise NO_BASE_BLOCK;
        (exit and terminate the Static Scheduler)

  end loop CREATE_BLOCKS;

end FIND_BASE_BLOCKS;
```

```
procedure FIND_BLOCK_LENGTH ( BASE_BLOCK : in out INTEGER;
                              HARMONIC_BLOCK_LENGTH : out INTEGER ) is
  NEW_GCD : INTEGER;
  NEW_LCM : INTEGER;
  LCM     : INTEGER;
  GCD     : INTEGER;
  ENTRY   : INTEGER;

function FIND_GCD ( ENTRY : in BASE_BLOCK ) returns NEW_GCD;
function FIND_LCM ( ENTRY : in BASE_BLOCK ) returns NEW_LCM;

begin
  GET ( ENTRY[i] : out BASE_BLOCK );
  GCD := ENTRY[i];
  LCM := ENTRY[i];
  GET ( ENTRY[i+1] : out BASE_BLOCK );
  while BASE_BLOCK not= {} loop

    function FIND_GCD returns NEW_GCD is
    begin
      while GCD not = 0 loop
        REMAINS1 := LCM mod GCD;
        REMAINS2 := ENTRY[i+1] mod GCD;
        if REMAINS1 = 0 and REMAINS2 = 0 then
          begin
            NEW_GCD := GCD;
            return NEW_GCD;
          end;
        else
          GCD := GCD - 1;
        end if;
      end loop;
    end FIND_GCD;

    function FIND_LCM returns NEW_GCD is
      begin
        NEW_LCM := ( LCM * ENTRY[i+1] / GCD );
        return NEW_LCM;
    end FIND_LCM;

     ENTRY[i] := ENTRY[i+1];
     GET ( ENTRY[i] : out BASE_BLOCK );
     LCM := NEW_LCM;
     GCD := NEW_GCD;
   end loop;
   HARMONIC_BLOCK_LENGTH := LCM;

 end FIND_BLOCK_LENGTH;

end HARMONIC_BLOCK_BUILDER;
```

```
with FILES;
with TEXT_IO;
package body OPERATOR_SCHEDULER is

  procedure TEST_DATA is
  begin

    procedure CALC_HALF_PERIODS ( OPERATORS : TEXT_IO.IN_FILE ) is

    function DIVIDE_PERIOD_BY_2 ( THE PERIOD : out OPERATORS ) returns
                                                          HALF_PERIODS;
    begin

    function DIVIDE_PERIOD_BY_2 returns HALF_PERIODS is
      HALF_PERIOD : REAL;
    begin
      while not OPERATORS.END_OF_FILE loop
        begin
          GET ( THE_PERIOD[i] : out OPERATORS );
          GET ( THE_MET[i] : out OPERATORS );
          HALF_PERIOD := THE_PERIOD[i] / 2;
          if HALF_PERIOD =< THE_MET[i] then

            exception
              raise FAIL_HALF_PERIOD;

          end if;
        end;
      end loop;
    end CALC_HALF_PERIODS;


  procedure CALC_TOTAL_TIME ( OPERATORS : TEXT_IO.IN_FILE;
                              HARMONIC_BLOCK_LENGTH : in INTEGER ) is
    TIMES      : REAL  := 0.0;
    OP_TIME    : REAL  := 0.0;
    TOTAL_TIME: REAL  := 0.0;

  function CALC_NO_OF_PERIODS ( HARMONIC_BLOCK_LENGTH : out OPERATORS;
                          THE_PERIOD : out OPERATORS ) returns TIMES;
  function MULTIPLY_BY_MET ( TIMES : in out REAL;
                           THE_MET : out OPERATORS ) returns OP_TIME;
  function ADD_TO_SUM ( OP_TIME : in out REAL ) returns TOTAL_TIME;

  begin
    while not OPERATORS.END_OF_FILE loop
      begin

        function CALC_NO_OF_PERIODS returns TIMES is
        begin
          GET ( THE_PERIOD[i] : in out OPERATORS );
          TIMES := HARMONIC_BLOCK_LENGTH / THE_PERIOD[i];
          return TIMES;
        end CALC_NO_OF_PERIODS;
```

```
      function MULTIPLY_BY_MET returns OP_TIME is
      begin
        GET ( THE_MET[i] : in out OPERATORS );
        OP_TIME := TIMES * THE_MET[i];
        return OP_TIME;
      end MULTIPLY_BY_MET;

      function ADD_TO_SUM returns TOTAL_TIME is
      begin
        TOTAL_TIME := TOTAL_TIME + OP_TIME;
        return TOTAL_TIME;
      end ADD_TO_SUM;

    end;
  end loop;

  exception
    when TOTAL_TIME > HARMONIC_BLOCK_LENGTH =>
      raise BAD_TOTAL_TIME;
      (exit and terminate the Static Schedule)

end CALC_TOTAL_TIME;



procedure CALC_RATIO_SUM ( OPERATORS : TEXT_IO.IN_FILE ) is

  RATIO      : REAL  := 0.0;
  RATIO_SUM  : REAL  := 0.0;

function DIVIDE_MET_BY_PERIOD ( THE_MET : in out OPERATORS;
                        THE_PERIOD : in out OPERATORS ) returns RATIO;
function ADD_TO_TIME ( RATIO : in out REAL ) returns RATIO_SUM;

begin
  while not OPERATORS.END_OF_FILE loop
    begin

      function DIVIDE_MET_BY_PERIOD returns RATIO is
      begin
        GET ( THE_MET[i] : out OPERATORS );
        GET ( THE_PERIOD[i] : out OPERATORS );
        RATIO := THE_MET / THE_PERIOD;
        return RATIO;
      end DIVIDE_MET_BY_PERIOD;

      function ADD_TO_TIME returns RATIO_SUM is
      begin
        RATIO_SUM := RATIO_SUM + RATIO;
        return RATIO_SUM;
      end ADD_TO_SUM;

    end;
  end loop;

  exception
```

```
      when RATIO_SUM > 0.5 =>
        raise RATIO_TOO_BIG;
        (exit and terminate the Static Scheduler)

  end CALC_RATIO_SUM;

end TEST_DATA;


  procedure VERIFY_TIME_LEFT ( HARMONIC_BLOCK_LENGTH : in INTEGER;
                               TIME : in INTEGER;
                               CONTINUE : BOOLEAN );
  begin
    if TIME >= HARMONIC_BLOCK_LENGTH then
      CONTINUE := FALSE;
    else

      exception
        raise OVER_TIME;
          (exit and terminate the Static Scheduler)

  end VERIFY_TIME_LEFT;


  procedure CREATE_INTERVAL ( THE_PERIODS : in OPERATORS;
                              SCHEDULE_INPUTS : TEXT_IO.OUT_FILE ) is

    LOWER_BOUND  : INTEGER  := 0;
    UPPER_BOUND  : INTEGER  := 0;

  function CALC_LOWER ( TIME : in out INTEGER;
                THE_PERIOD : in out OPERATORS ) returns LOWER_BOUND;
  function CALC_UPPER ( TIME : in out INTEGER;
                THE_PERIOD : in out OPERATORS;
                THE_MET : in out OPERATORS ) returns UPPER_BOUND;

  begin

    function CALC_LOWER returns LOWER_BOUND is
    begin
      GET ( THE_START[i] : in out SCHEDULE_INPUTS );
      GET ( THE_PERIOD[i] : in out OPERATORS );
      LOWER_BOUND[i] := THE_START[i] + THE_PERIOD[i];
      returns LOWER_BOUND;
    end CALC_LOWER;

    PUT ( LOWER_BOUND[i] : out SCHEDULE_INPUTS.THE_LOWER[i] );

    function CALC_UPPER returns UPPER_BOUND is
    begin
      GET ( THE_START[i] : in out PRECEDENCE_LIST );
      GET ( THE_PERIOD[i] : in out OPERATORS );
      GET ( THE_MET[i] : in out OPERATORS );
      UPPER_BOUND[i] := [THE_START[i] + (2*THE_PERIOD[i]) - THE_MET[i] ];
```

```
    returns UPPER_BOUND;
  end CALC_UPPER;

  PUT ( UPPER_BOUND[i] : out SCHEDULE_INPUTS.THE_UPPER[i] );

end CREATE_INTERVAL;


procedure SCHEDULE_INITIAL_SET ( PRECEDENCE_LIST : TEXT_IO.IN_FILE;
                                 SCHEDULE_INPUTS : TEXT_IO.OUT_FILE;
                                 CONTINUE : in out BOOLEAN;
                                 THE_MET : in out OPERATORS ) is

  START_TIME : INTEGER  := 0;
  STOP_TIME  : INTEGER  := 0;

begin
  procedure VERIFY_TIME_LEFT ( CONTINUE : in out BOOLEAN );
  begin
    while not PRECEDENCE_LIST.END_OF_FILE loop
      GET (THE_LEFT_OP_ID[i] : in out PRECEDENCE_LIST );
      if THE_LEFT_OP_ID not= EXTERNAL then
        begin
          PUT ( THE_LEFT_OP_ID : in SCHEDULE_INPUTS.OPERATOR[i] );
          PUT ( START_TIME : in SCHEDULE_INPUTS.THE_START[i] );
          GET ( THE_MET[i] : in out OPERATOR );
          STOP_TIME[i] := START_TIME[i] + THE_MET[i];
          PUT ( THE STOP_TIME[i] : in SCHEDULE_INPUTS.THE_STOP[i] );
        end;
      else
        restart with THE_LEFT_OP_ID[i+1];
      end if;
      START_TIME := STOP_TIME[i];
      procedure CREATE_INTERVAL;
    end loop;
  end;

end SCHEDULE_INITIAL_SET;



procedure SCHEDULE_REST_OF_BLOCK (SCHEDULE_INPUTS : TEXT_IO.OUT_FILE;
                                  CONTINUE : in out BOOLEAN;
                                  THE_MET : in out OPERATORS ) is

begin
  procedure VERIFY_TIME_LEFT ( CONTINUE : in out BOOLEAN );
  while not SCHEDULE_INPUTS.END_OF_FILE loop
    GET ( THE_LOWER.'SMALLEST : in out SCHEDULE_INPUTS );
    GET ( THE_STOP.'LARGEST : in out SCHEDULE_INPUTS );
    if THE_LOWER.'SMALLEST >= THE_STOP.'LARGEST then
      begin
        START_TIME := THE_LOWER.'SMALLEST;
        GET (OPERATOR.'SMALLEST : in out SCHEDULE_INPUTS );
        PUT ( START_TIME in SCHEDULE_INPUTS.THE_START );
        GET ( THE_MET : in out OPERATORS );
```

```
        STOP_TIME := START_TIME + THE_MET;
        PUT ( STOP_TIME : in SCHEDULE_INPUTS.THE_STOP );
      end;
    else

      exception
        when others =>
          raise INVALID_SCHEDULE;
          (exit and terminate the Static Scheduler)

    end if;
    START_TIME := STOP_TIME;
    procedure CREATE_INTERVAL;
  end loop;

end SCHEDULE_REST_OF_BLOCK;

procedure CREATE_STATIC_SCHEDULE ( SCHEDULE_INPUTS : TEXT_IO.IN_FILE;
                                   STATIC_SCHEDULE : TEXT_IO.OUT_FILE ) is

begin
  CREATE ( STATIC_SCHEDULE : TEXT_IO.OUT_FILE );
  PUT_LINE ("package THE_STATIC_SCHEDULE is ");
  PUT_LINE ("task SEQUENCE_OF_CALLS is ");
  while not SCHEDULE_INPUTS.END_OF_FILE loop
    begin
      GET ( OPERATOR[i] : in out SCHEDULE_INPUTS );
      PUT ("entry EACH_OPERATOR." OPERATOR[i] );
      GET ( THE_START[i] : in out SCHEDULE_INPUTS );
      PUT_LINE ( "("THE_START[i]" : in out INTEGER;" );
      GET ( THE_STOP[i] : in out SCHEDULE_INPUTS );
      PUT_LINE ( THE_STOP[i]" : in out INTEGER);" );
      GET ( THE_START[i+1] : in out SCHEDULE_INPUTS );
      if THE_START[i+1] = THE_STOP[i] then
        begin
          OPERATOR[i] := OPERATOR[i+1];
          return to the beginning of the loop
        end;
      elsif
        THE_START[i+1] > THE_STOP[i] then
        begin
          PUT_LINE ("entry DYNAMIC ("THE_STOP[i]" : in out INTEGER;" );
          PUT_LINE (THE_START[i+1] " : in out INTEGER );" );
          OPERATOR[i] := OPERATOR[i+1];
          return to beginning of the loop;
        end;
      else

        exception
          when THE_START[i+1] < THE_STOP[i] =>
            raise SCHEDULE_ERROR;
            (exit and terminate the Static Scheduler)

      end if;
    end;
  end loop;
```

```
      PUT_LINE ( "end SEQUENCE_OF_CALLS;" );

    while not SCHEDULE_INPUTS.END_OF_FILE loop       -- second time for body
      PUT_LINE ( "with CALENDAR;" );
      PUT_LINE ( "package body THE_STATIC_SCHEDULE is" );
      PUT_LINE ( "begin" );
      PUT_LINE ( "task body SEQUENCE_OF_CALLS is" );
      PUT_LINE ( "begin" );
      PUT_LINE ( "loop" );
      PUT_LINE ( "procedure THE_OPERATOR is" );
      PUT_LINE ( "begin" );
      PUT ("accept EACH_OPERATOR." OPERATOR[i] );
      PUT_LINE ( " (" THE_START[i]" : in out INTEGER;" );
      PUT_LINE ( THE_STOP[i] " : in out INTEGER ) do" );
      PUT_LINE ( "select" );
      PUT_LINE ( "when CLOCK.VALUE < " THE_STOP[i] " then" );
      PUT_LINE ( "entry DYNAMIC" );
      PUT_LINE ( "or" );
      PUT_LINE ( "when CLOCK.VALUE = " THE_STOP[i] " then" );
      PUT_LINE ( "entry EACH_OPERATOR." OPERATOR[i+1] );
      PUT_LINE ( "else" );
      PUT_LINE ( "exception" );
      PUT_LINE ( "when others =>" );
      PUT_LINE ( "raise RUNTIME_MET_FAILURE" );
      PUT_LINE ( "end THE_OPERATOR;" );
      PUT_LINE ( "end loop;" );
      PUT_LINE ( "end SEQUENCE_OF_CALLS;" );
      PUT_LINE ( "end THE_STATIC_SCHEDULE;" );

    end OPERATOR_SCHEDULER;

end STATIC_SCHEDULER;
```

APPENDIX F. LIST OF ACRONYMS

AG	Attribute Grammar
CAPS	Computer Aided Prototyping System
CLE	Combiner Linker Exporter
DFD	Data Flow Diagram
DOD	Department of Defense
DON	Department of the Navy
ESS	Execution Support System
FIFO	First-In First-Out
GCD	Greatest Common Denominator
I/O	Input/Output
LCM	Least Common Multiple
MCP	Minimum Calling Period
MET	Maximum Execution Time
MRT	Maximum Response Time
PSDL	Prototype System Description Language
SDMS	Software Design Management System
SECNAV	Secretary of the Navy
SS/TDMA	Satellite-Switched Time Division Multiple Access
TDMA	Time Division Multiple Access
UI	User Interface

LIST OF REFERENCES

1. Booch, G., *Software Engineering with Ada®*, 2nd ed., Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1987.

2. Jensen, E., Locke, C., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems", *IEEE Proceedings of the Real-Time Systems Symposium*, San Diego, CA, pp. 112-122, December 3-6, 1985.

3. Luqi and Ketabchi, M., *A Computer Aided Prototyping System*, Technical Report NPS52-87-011, Naval Postgraduate School, Monterey, CA, 1987 and in *IEEE Software*, pp. 66-72, March 1988.

4. Luqi, *Execution of Real-Time Prototypes*, Technical Report NPS52-87-012, Naval Postgraduate School, Monterey, CA, 1987 and in *ACM First International Workshop on Computer-Aided Software Engineering*, Cambridge, MA, Vol. 2, pp. 870-884, May 1987.

5. Luqi, *Research Aspects of Rapid Prototyping*, Technical Report NPS52-87-006, Naval Postgraduate School, Monterey, CA, 1987.

6. Luqi, *Rapid Prototyping for Large Software System Design*, Ph.D. thesis, University of Minnesota, Duluth, MN, May 1986.

7. Luqi and Berzins, V., "Languages for Specification, Design, and Prototyping" chapter in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

8. Luqi, *Normalized Specifications for Identifying Reusable Software*, Technical Report NPS52-87-007, Naval Postgraduate School, Monterey, CA, 1987 and in *Proceedings of the ACM-IEEE Fall Joint Computer Conference*, Dallas, TX, pp. 46-49, October 1987.

9. Luqi, Berzins, V., and Yeh, R., "A Prototyping Language for Real Time Software", to appear in *IEEE Transactions on Software Engineering*, 1988.

10. Mok, A., "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proceedings of the Real-Time Systems Symposium*, Austin, TX, pp. 125-134, December 4-6, 1984.

11. Herndon, R., *The Incompleat AG User's Guide and Reference Manual*, Technical Report 85-37, University of Minnesota, Minneapolis, MN, 1985.

12. Moffitt, C., *A Language Translator for a Computer Aided Rapid Prototyping System*, M.S. thesis, Naval Postgraduate School, Monterey, CA, March 1988.

13. Luqi and Berzins, V., *Rapid Prototyping of Real-Time Systems*. Technical Report NPS52-87-005, Naval Postgraduate School, Monterey, CA, 1987.

14. Eaton, S., *A Dynamic Scheduler for the Computer Aided Prototyping System (CAPS)*, M.S. thesis, Naval Postgraduate School, Monterey, CA, March 1988.

15. O'Hern, J., *A Conceptualized Level Design for a Static Scheduler for Hard Real-Time Systems*, M.S. thesis, Naval Postgraduate School, Monterey, CA, March 1988.

16. Mok, A., "The Design of Real-Time Programming Systems Based on Process Models", *IEEE Proceedings of the Real-Time Systems Symposium*, Austin, TX, pp. 5-17, December 4-6, 1984.

17. Stallings, W., *Data and Computer Communications*, Macmillan Publishing Co., New York, NY, 1985.

18. Department of the Navy, SECNAV INSTRUCTION 5200.37, *Acquisition of Software-Intensive C2 Information Systems*, January 5, 1988.

BIBLIOGRAPHY

Abbott, C., "Intervention Schedules for Real-Time Programming", *IEEE Transactions on Software Engineering*, Vol. SE-10 No. 3, pp. 268-274, May 1984.

Cheng, S., Stankovic, J., and Ramamrithamm, K., "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems", *IEEE Proceedings of the Real-Time Systems Symposium* New Orleans, LA, pp. 175-180, December 2-4, 1986.

Dasarathy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", *Proceedings of the Real-Time Systems Symposium*, Los Angeles, CA, pp. 197-204, December 7-9, 1982.

Jahanian, F. and Mok, A., "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. SE-12 No. 9, pp. 890-904, September 1986.

Levenson, N., *Building Safe Software*, Technical Report No. 86-14, University of California at Irvine, February 1986.

Mok, A., "A Graph-Based Computation Model for Real-Time Systems", *IEEE Proceedings of the International Conference on Parallel Processing*, Pennsylvania State University, PA, pp. 619-623, August 20-23, 1985.

Mok, A. and Sutanthavibul, S., "Modeling and Scheduling of Dataflow Real-Time Systems", *IEEE Proceedings of the Real-Time Systems Symposium*, San Diego, CA, pp. 178-187, December 3-6, 1985.

Plattner, B., "Real-Time Execution Monitoring", *IEEE Transactions on Software Engineering*, Vol. SE-10 No. 6, pp. 756-764, November 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	LT Dorothy M. Janson, USN USCINCEUR Headquarters General Delivery APO New York, NY 09128-4209	2
4.	Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	2
5.	Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20350	2
6.	Commander Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue N.W. Washington, D.C. 20390-5290	2
7.	Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20397-5340	1
8.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, D.C. 20363-5100	1
9.	Ada® Joint Program Office OUSDRE(R&AT) The Pentagon Washington, D.C. 20301	1
10.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	2

11. Professor Luqi
Code 52LQ
Naval Postgraduate School
Monterey, CA 93943 — 1

12. Professor Taracad Sivasankaran
Code 54SJ
Naval Postgraduate School
Monterey, CA 93943 — 1

13. Professor D. C. Boger
Code 54BO
Naval Postgraduate School
Monterey, CA 93943 — 1

14. Defense Communications Agency
Attn: LT S.L. Eaton, USN, Code B531
Washington, D.C. 20305 — 1

15. LT Charlie R. Moffitt, II, USN
Department Head Class # 104
SWOSCOLCOM, Bldg. 446
Newport, RI 02841-5012 — 1

16. Commander, Naval Security Group Command
Attn: LT Joanne T. O'Hern, Code G30
3801 Nebraska Avenue, N.W.
Washington, D.C. 20390 — 1

17. Naval Sea Systems Command
Attn. CAPT Joel Crandall
National Center # 2, Suite 7N06
Washington, D.C. 22202 — 1

18. Office of the Secretary of Defense
Attn. CDR Barber
The Star Program
Washington, D.C. 20301 — 1

19. Navy Ocean System Center
Attn. Lindwood Sutton, Code 423
San Diego, CA 92152-5000 — 1

20. RADC/COES
Attn. LT Kevin Benner
Griffiss Air Force Base
New York, NY 13441-5700 — 1

21. MAJ Mike Dolezal
Director, Development Center
MCDEC
Quantico, VA 22134-5080 — 1

Thesis
J2912
c.1

Janson
A static schedular for the computer aided prototyping system: an implementation guide.